A Theory of Skiplists with Applications to the Verification of Concurrent Datatypes*

Alejandro Sánchez[1] and César Sánchez[1,2]

[1] The IMDEA Software Institute, Madrid, Spain
[2] Spanish Council for Scientific Research (CSIC), Spain
{alejandro.sanchez,cesar.sanchez}@imdea.org

Abstract. This paper presents a theory of skiplists with a decidable satisfiability problem, and shows its applications to the verification of concurrent skiplist implementations. A skiplist is a data structure used to implement sets by maintaining several ordered singly-linked lists in memory, with a performance comparable to balanced binary trees. We define a theory capable of expressing the memory layout of a skiplist and show a decision procedure for the satisfiability problem of this theory. We illustrate the application of our decision procedure to the temporal verification of an implementation of concurrent lock-coupling skiplists. Concurrent lock-coupling skiplists are a particular version of skiplists where every node contains a lock at each possible level, reducing granularity of mutual exclusion sections.

The first contribution of this paper is the theory $\mathsf{TSL_K}$. $\mathsf{TSL_K}$ is a decidable theory capable of reasoning about list reachability, locks, ordered lists, and sublists of ordered lists. The second contribution is a proof that $\mathsf{TSL_K}$ enjoys a finite model property and thus it is decidable. Finally, we show how to reduce the satisfiability problem of quantifier-free $\mathsf{TSL_K}$ formulas to a combination of theories for which a many-sorted version of Nelson-Oppen can be applied.

1 Introduction

A skiplist [14] is a data structure that implements sets, maintaining several sorted singly-linked lists in memory. Skiplists are structured in multiple levels, where each level consists of a single linked list. The skiplist property establishes that the list at level $i+1$ is a sublist of the list at level i. Each node in a skiplist stores a value and at least the pointer corresponding to the lowest level list. Some nodes also contain pointers at higher levels, pointing to the next element present at that level. The advantage of skiplists is that they are simpler and more efficient to implement than search trees, and search is still (probabilistically) logarithmic.

* This work was funded in part by the EU project FET IST-231620 *HATS*, MICINN project TIN-2008-05624 *DOVES*, CAM project S2009TIC-1465 *PROMETIDOS*, and by the COST Action IC0901 *Rich ModelToolkit-An Infrastructure for Reliable Computer Systems.*

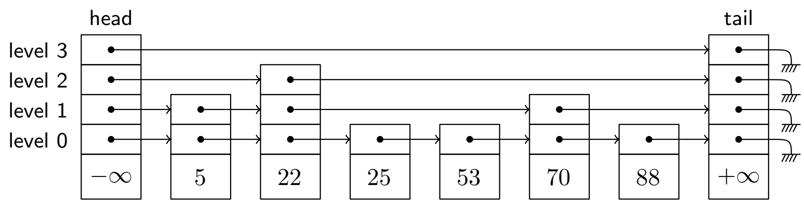


Fig. 1. A skiplist with 4 levels

Consider the skiplist shown in Fig. 1. Contrary to single-linked lists implementations, higher-level pointers allow to *skip* many elements during the search. A search is performed from left to right in a top down fashion, progressing as much as possible in a level before descending. For instance, in Fig. 1 a search for value 88 starts at level 3 of node *head*. From *head* the pointer at level 3 reaches *tail* with value $+\infty$, which is greater than 88. Hence the search algorithm moves down one level at *head* to level 2. The successor at level 2 contains value 22, which is smaller than 88, so the search continues at level 2 until a node containing a greater value is found. At that moment, the search moves down one further level again. The expected logarithmic search follows from the probability of any given node occurs at a certain level decreasing by $1/2$ as a level increases (see [14] for an analysis of the running time of skiplists).

We are interested in the formal verification of implementations of skiplists, in particular in temporal verification (liveness and safety properties) of sequential and concurrent implementations. This verification activity requires to deal with unbounded mutable data. One popular approach to verification of heap programs is Separation Logic [17]. Skiplists, however, are problematic for separation-like approaches due to the aliasing and memory sharing between nodes at different levels. Based on the success of separation logic some researchers have extended this logic to deal with concurrent programs [23, 7], but concurrent datatypes follow a programming style in which the activities of concurrent threads are not structured according to critical regions with memory footprints. In these approaches based on Separation Logic memory regions are implicitly declared (hidden in the separation conjunction), which makes the reasoning about unstructured concurrency more cumbersome.

Most of the work in formal verification of pointer programs follows program logics in the Hoare tradition, either using separation logic or with specialized logics to deal with the heap and pointer structures [9, 24, 3]. However, extending these logics to deal with concurrent programs is hard, and though some success has been accomplished it is still an open area of research, particularly for liveness.

Continuing our previous work [18] we follow a complementary approach. We start from temporal deductive verification in the style of Manna-Pnueli [11], in particular using general verification diagrams [5, 19] to deal with concurrency. This style of reasoning allows a clean separation in a proof between the temporal part (why the interleavings of actions that a set of threads can perform satisfy a certain property) with the underlying data being manipulated. A verification diagram decomposes a formal proof into a finite collection of verification conditions (VC), each of which corresponds to the effect that a small step in the program has in the data. To automatize the process of checking the proof represented by a verification diagram it is necessary to use decision procedures for the kind of data structures manipulated. This paper studies the automatic verification of VCs for the case of skiplists.

Logics like [9, 24, 3] are very powerful to describe pointer structures, but they require the use of quantifiers to reach their expressive power. Hence, these logics preclude a combination a-la Nelson-Oppen [12] or BAPA [8] with other aspects of the program state. Instead, our solution starts from a quantifier-free theory of single-linked lists [16], and extends it in a non trivial way with order and sublists of ordered lists. The logic obtained can express skiplist-like properties without using quantifiers, allowing the combination with other theories. Proofs for an unbounded number of threads are achieved by parameterizing verification diagrams, splitting cases for interesting threads and producing a single verification condition to generalize the remaining cases. However, in this paper we mainly focus in the decision procedure. Since we want to verify concurrent lock-based implementations we extend the basic theory with locks, lock ownership, and sets of locks (and in general stores of locks). The decision procedure that we present here supports the manipulation of explicit regions, as in regional logic [2] equipped with *masked regions*, which enables reasoning about disjoint portions of the same memory cell. We use masked regions to "separate" different levels of the same skiplist node.

We call our theory TSL_K, that allows to reason about skiplists of height at most K. To illustrate the use of this theory, we sketch the proof of termination of every invocation of an implementation of a lock-coupling concurrent skiplist.

The rest of the paper is structured as follows. Section 2 presents lock-coupling concurrent skiplists. Section 3 introduces TSL_K. Section 4 shows that TSL_K is decidable by proving a finite model property theorem, and describes how to construct a more efficient decision procedure using the many-sorted Nelson-Oppen combination method. Finally, Section 5 concludes the paper. Some proofs are missing due to space limitation.

2 Fine-Grained Concurrent Lock-Coupling Skiplists

In this section we present a simple concurrent implementation of skiplists that uses lock-coupling [6] to acquire and release locks. This implementation can be seen as an extension of concurrent lock-coupling lists [6, 23] to multiple layers of pointers. This algorithm imposes a locking discipline, consisting of acquiring locks as the search progresses, and releasing a node's lock only after the lock of the next node in the search process has been acquired. A naïve implementation of this solution would equip each node with a single lock, allowing multiple threads to access simultaneously different nodes in the list, but protecting concurrent accesses to two different fields of the same node. The performance can be improved by carefully allowing multiple threads to simultaneously access the same node at different levels. We study here an implementation of this faster solution in which each node is equipped with a different lock at each level. At execution time a thread uses locks to protect the access to only some fields of a given node. A precise reasoning framework needs to capture those portions of the memory protected by a set of locks, which may include only *parts* of a node. Approaches based on strict separation (separation logic [17] or regional logic [2]) do not provide the fine grain needed to reason about individual fields of shared objects. Here, we introduce the concept of *masked regions* to describe regions and the fields within. A masked region consists of a set of pairs formed by a region (*Node* cell) and a field (a skiplist level): $\mathbf{mrgn} \mathrel{\hat{=}} 2^{Node \times \mathbb{N}}$ We call the field a mask, since it identifies which part of the object is relevant. For example, in Fig. 2 the region within dots represents the area of the memory that thread j is protecting. This portion of the memory is described by the masked region $\{(n_2, 2), (n_5, 2), (n_2, 1), (n_4, 1), (n_3, 0), (n_4, 0)\}$. As with regional logic, an empty set intersection denotes separation. In masked regions two memory nodes at different levels do not overlap. This notion is similar to data-groups [10].

Fig. 3(a) contains the pseudo-code declaration of the *Node* and *SkipList* classes. Throughout the paper we use `//@` to denote ghost code added for verification purposes. Note that the structure is parametrized by a value K, which determines the maximum possible level of any node in the modeled skiplist. The fields *val* and *key* in the class *Node* contains the value and the key of the element used to order them. Then, we can store key-value pairs, or use the skiplist as a set of arbitrary elements as long as the key can be used to compare. The *next* array stores the pointers to the next nodes at each of the possible K different levels of the skiplist. Finally, the *lock* array keeps the locks, one for each level, protecting the access to the corresponding *next* field. The *SkipList* class contains two pointer fields: *head* and *tail* plus a ghost variable field r. Field *head* points to the first node of the skiplist, and *tail* to the last one. Variable r, only used for verification purposes, keeps the (masked) region represented by all nodes in the skiplist with all their levels. In this implementation, *head* and *tail* are sentinel nodes, with $key = -\infty$ and $key = +\infty$, respectively. For simplicity, these nodes are not eliminated during the execution and their *val* field remains unchanged.

Fig. 3(b) shows the implementation of the insertion algorithm. The algorithms for searching and removing are similar, and omitted due to space limitations. The ghost variable m_r stores a masked region containing all the nodes

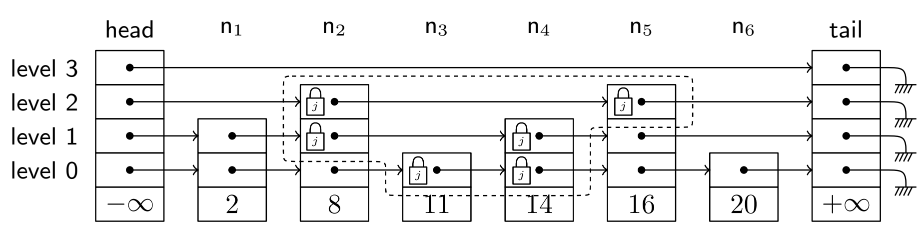


Fig. 2. A skiplist with the masked region given by the fields locked by thread j

```
class Node {                                class SkipList {
    Value val;                                  Node* head;
    Key key;                                    Node* tail;
    Array⟨Node*⟩(K) next;                       //@ mrgn r;
    Array⟨Node*⟩(K) lock;                   }
}
```

(a) data structures

```
procedure Insert(SkipList sl, Value newval)
    Vector⟨Node*⟩ upd[0..K − 1]                  //@ mrgn m_r := ∅
    lvl := randomLevel(K)
    Node* pred := sl.head
    pred.locks[K − 1].lock()                     //@ m_r := m_r ∪ {(pred, K − 1)}
    Node* curr := pred.next[K − 1]
    curr.locks[K − 1].lock()                     //@ m_r := m_r ∪ {(curr, K − 1)}
    for i := K − 1 downto 0 do
        if i < K − 1 then
            pred.locks[i].lock()                 //@ m_r := m_r ∪ {(pred, i)}
            curr := pred.next[i]
            curr.locks[i].lock()                 //@ m_r := m_r ∪ {(curr, i)}
            if i ≥ lvl then
                curr.locks[i + 1].unlock()       //@ m_r := m_r − {(curr, i + 1)}
                pred.locks[i + 1].unlock()       //@ m_r := m_r − {(pred, i + 1)}
            end if
        end if
        while curr.val < newval do
            pred.locks[i].unlock()               //@ m_r := m_r − {(pred, i)}
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()                 //@ m_r := m_r ∪ {(curr, i)}
        end while
        upd[i] := pred
    end for
    Bool valueWasIn := (curr.val = newval)
    if valueWasIn then
        for i := 0 to lvl do
            upd[i].next[i].locks[i].unlock()     //@ m_r := m_r − {(upd[i].next[i], i)}
            upd[i].locks[i].unlock()             //@ m_r := m_r − {(upd[i], i)}
        end for
    else
        x := CreateNode(lvl, newval)
        for i := 0 to lvl do
            x.next[i] := upd[i].next[i]
            upd[i].next[i] := x                  //@ sl.r := sl.r ∪ {(x, i)}
            x.next[i].locks[i].unlock()          //@ m_r := m_r − {(x.next[i], i)}
            upd[i].locks[i].unlock()             //@ m_r := m_r − {(upd[i], i)}
        end for
    end if
    return ¬valueWasIn
end procedure
```

(b) insertion algorithm

Fig. 3. Data structure and insert algorithm for concurrent lock-coupling skiplist

and fields currently locked by the running thread. The set operations $\cup$ and $-$ are used for the manipulation of the corresponding sets of pairs.

Let sl be a pointer to a skiplist (an instance of the class described in Fig. 3(a)). The following predicate captures whether sl points to a well-formed skiplist of height 4 or less:

$$\begin{aligned}
\mathit{SkipList}_4(h, sl : \mathit{SkipList}) \;\hat{=}\; & \mathit{OList}(h, sl, 0) \;\wedge & (1)\\
& \begin{pmatrix} h[sl].\mathit{tail}.\mathit{next}[0] = \mathit{null} \wedge h[sl].\mathit{tail}.\mathit{next}[1] = \mathit{null} \\ h[sl].\mathit{tail}.\mathit{next}[2] = \mathit{null} \wedge h[sl].\mathit{tail}.\mathit{next}[3] = \mathit{null} \end{pmatrix} \wedge & (2)\\
& \begin{pmatrix} \mathit{SubList}(h, sl.\mathit{head}, sl.\mathit{tail}, 1, sl.\mathit{head}, sl.\mathit{tail}, 0) \wedge \\ \mathit{SubList}(h, sl.\mathit{head}, sl.\mathit{tail}, 2, sl.\mathit{head}, sl.\mathit{tail}, 1) \wedge \\ \mathit{SubList}(h, sl.\mathit{head}, sl.\mathit{tail}, 3, sl.\mathit{head}, sl.\mathit{tail}, 2) \end{pmatrix} & (3)
\end{aligned}$$

The predicate OList in (1) describes that in heap h, the pointer sl is an ordered linked-lists when repeatedly following the pointers at level 0 starting at head. The predicate (2) indicates all levels are null terminated, and (3) indicates that each level is in fact a sublist of its nearest lower level. Predicates of this kind also allow to express the effect of programs statements via first order transition relations. Consider the statement at line 36 in program insert shown in Fig. 3(b) on a skiplist of height 4, taken by thread with id t. This transition corresponds to a new node x at level i being connected to the skiplist. If the memory layout from pointer sl is that of a skiplist before the statement at line 36 is executed, then it is also a skiplist after the execution:

$$\mathit{SkipList}_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \rightarrow \mathit{SkipList}_4(h', sl')$$

The effect of the statement at line 36 is represented by the first-order transition relation $\rho_{36}^{[t]}$. To ensure this property, i is required to be a valid level, and the key of the nodes that will be pointing to x must be lower than the key of node x. Moreover, the masked region of locked nodes remains unchanged. Predicate φ_{aux} contains support invariants. For simplicity, we use prev for $\mathit{upd}^{[t]}[i]$. Then, the full verification condition is:

$$\mathit{SkipList}_4(h, sl) \wedge \begin{pmatrix} x.\mathit{key} = \mathit{newval} & \wedge \\ \mathit{prev}.\mathit{key} < \mathit{newval} & \wedge \\ x.\mathit{next}[i].\mathit{key} > \mathit{newval} & \wedge \\ \mathit{prev}.\mathit{next}[i] = x.\mathit{next}[i] & \wedge \\ (x, i) \notin sl.r \wedge 0 \le i \le 3 & \end{pmatrix} \wedge \begin{pmatrix} at_{36}[t] & \wedge \\ \mathit{prev}'.\mathit{next}[i] = x & \wedge \\ at'_{37}[t] & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ x' = x & \ldots \end{pmatrix} \rightarrow$$

$$\mathit{SkipList}_4(h', sl')$$

As usual, we use primed variables to describe the values of the variables after the transition is taken. Section 4 contains a full verification condition. This example illustrates that to be able to automatically prove VCs for the verification of skiplist manipulating algorithms, we require a theory that allows to reason about heaps, addresses, nodes, masked regions, ordered lists and sublists.

3 The Theory of Concurrent Skiplists of Height K: $\mathsf{TSL_K}$

We build a decision procedure to reason about skiplist of height K combining different theories, aiming to represent pointer data structures with a skiplist layout, masked regions and locks. We extend the Theory of Concurrent Linked Lists ($\mathsf{TLL3}$) [18], a decidable theory that includes reachability of concurrent list-like structures in the following way:

- each node is equipped with a *key* field, used to reason about element's order.
- the reasoning about single level lists is extended to all the K levels.
- we extend the theory of regions with masked regions.
- lists are extended to ordered lists and sub-paths of ordered lists.

We begin with a brief description of the basic notation and concepts. A signature Σ is a triple (S, F, P) where S is a set of sorts, F a set of functions and P a set of predicates. If $\Sigma_1 = (S_1, F_1, P_1)$ and $\Sigma_2 = (S_2, F_2, P_2)$, we define $\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2, P_1 \cup P_2)$. Similarly we say that $\Sigma_1 \subseteq \Sigma_2$ when $S_1 \subseteq S_2$, $F_1 \subseteq F_2$ and $P_1 \subseteq P_2$. If $t(\varphi)$ is a term (resp. formula), then we denote with $V_\sigma(t)$ (resp. $V_\sigma(\varphi)$) the set of variables of sort σ occurring in t (resp. φ).

A Σ-interpretation is a map from symbols in Σ to values. A Σ-structure is a Σ-interpretation over an empty set of variables. A Σ-formula over a set X of variables is satisfiable whenever it is true in some Σ-interpretation over X. Let Ω be a signature, $\mathcal{A}$ an Ω-interpretation over a set $V$of variables, $\Sigma \subseteq \Omega$ and $U \subseteq V$. $\mathcal{A}^{\Sigma,U}$ denotes the interpretation obtained from $\mathcal{A}$ restricting it to interpret only the symbols in Σ and the variables in U. We use $\mathcal{A}^{\Sigma}$ to denote $\mathcal{A}^{\Sigma,\emptyset}$. A Σ-theory is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class of Σ-structures. Given a theory $T = (\Sigma, \mathbf{A})$, a T-interpretation is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}^{\Sigma} \in \mathbf{A}$. Given a Σ-theory T, a Σ-formula φ over a set of variables X is T-satisfiable if it is true on a T-interpretation over X. Formally, the theory of skiplists of height K is defined as $\mathsf{TSL_K} = (\Sigma_{\mathsf{TSL_K}}, \mathbf{TSLK})$, where

$$\Sigma_{\mathsf{TSL_K}} = \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{reach}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{bridge}}$$

The signature of $\mathsf{TSL_K}$ is shown in Fig. 4. **TSLK** is the class of $\Sigma_{\mathsf{TSL_K}}$-structures satisfying the conditions depicted in Fig. 5. The symbols of Σ_{set} and Σ_{setth} follow their standard interpretation over sets of addresses and thread identifiers resp.

Informally, sort addr represents addresses; elem the universe of elements that can be stored in the skiplist; ord the ordered keys used to preserve a strict order in the skiplist; thid thread identifiers; $\mathsf{level_K}$ the levels of a skiplist; cell models *cells* representing a node in a skiplist; mem models the heap, mapping addresses to cells or to *null*; path describes finite sequences of non-repeating addresses to model non-cyclic list paths; set models sets of addresses – also known as regions –, while setth models sets of thread identifiers and mrgn masked regions.

$\Sigma_{\mathsf{level_K}}$ contains symbols for level identifiers $0, 1, \dots, \mathsf{K}-1$ and their conventional order. Σ_{ord} contains two special elements $-\infty$ and ∞ for the lowest and highest values in the order $\preceq$. Σ_{thid} only contains, besides $=$ and $\neq$ as for all the other theories, a special constant $\oslash$ to represent the absence of a thread identifier. Σ_{cell} contains the constructors and selectors for building and inspecting cells, including *error* for incorrect dereferences. Σ_{mem} is the signature for heaps, with the usual memory access and single memory mutation functions. Σ_{set} and Σ_{setth} are theories of sets of addresses and thread ids resp. Σ_{mrgn} is the theory of masked regions. The signature Σ_{reach} contains predicates to check reachability of address using paths at different levels, while Σ_{bridge} contains auxiliary functions and predicates to manipulate and inspect paths and locks.

Signt	Sort	Functions	Predicates
$\Sigma_{\mathsf{level_K}}$	$\mathsf{level_K}$	$0, 1, \ldots, \mathsf{K}-1 : \mathsf{level_K}$	$<: \mathsf{level_K} \times \mathsf{level_K}$
Σ_{ord}	ord	$-\infty, +\infty : \mathsf{ord}$	$\preceq : \mathsf{ord} \times \mathsf{ord}$
Σ_{thid}	thid	$\oslash : \mathsf{thid}$	
Σ_{cell}	cell elem ord addr thid	$\mathit{error} : \mathsf{cell}$ $\mathit{mkcell} : \mathsf{elem} \times \mathsf{ord} \times \mathsf{addr}^{\mathsf{K}} \times \mathsf{thid}^{\mathsf{K}} \to \mathsf{cell}$ $\_.\mathit{data} : \mathsf{cell} \to \mathsf{elem}$ $\_.\mathit{key} : \mathsf{cell} \to \mathsf{ord}$ $\_.\mathit{next}[\_] : \mathsf{cell} \times \mathsf{level_K} \to \mathsf{addr}$ $\_.\mathit{lockid}[\_] : \mathsf{cell} \times \mathsf{level_K} \to \mathsf{thid}$ $\_.\mathit{lock}[\_] : \mathsf{cell} \times \mathsf{level_K} \to \mathsf{thid} \to \mathsf{cell}$ $\_.\mathit{unlock}[\_] : \mathsf{cell} \times \mathsf{level_K} \to \mathsf{cell}$	
Σ_{mem}	mem addr cell	$\mathit{null} : \mathsf{addr}$ $\_[\_] : \mathsf{mem} \times \mathsf{addr} \to \mathsf{cell}$ $\mathit{upd} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{cell} \to \mathsf{mem}$	
Σ_{reach}	mem addr path	$\epsilon : \mathsf{path}$ $[\_] : \mathsf{addr} \to \mathsf{path}$	$\mathit{append} : \mathsf{path} \times \mathsf{path} \times \mathsf{path}$ $\mathit{reach}_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \times \mathsf{path}$
Σ_{set}	addr set	$\emptyset : \mathsf{set}$ $\{\_\} : \mathsf{addr} \to \mathsf{set}$ $\cup, \cap, \setminus : \mathsf{set} \times \mathsf{set} \to \mathsf{set}$	$\in : \mathsf{addr} \times \mathsf{set}$ $\subseteq : \mathsf{set} \times \mathsf{set}$
Σ_{setth}	thid setth	$\emptyset_T : \mathsf{setth}$ $\{\_\}_T : \mathsf{thid} \to \mathsf{setth}$ $\cup_T, \cap_T, \setminus_T : \mathsf{setth} \times \mathsf{setth} \to \mathsf{setth}$	$\in_T : \mathsf{thid} \times \mathsf{setth}$ $\subseteq_T : \mathsf{setth} \times \mathsf{setth}$
Σ_{mrgn}	mrgn addr $\mathsf{level_K}$	$\mathbf{emp}_{\mathsf{mr}} : \mathsf{mrgn}$ $\langle\_, \_\rangle_{\mathsf{mr}} : \mathsf{addr} \times \mathsf{level_K} \to \mathsf{mrgn}$ $\cup_{\mathsf{mr}}, \cap_{\mathsf{mr}}, -_{\mathsf{mr}} : \mathsf{mrgn} \times \mathsf{mrgn} \to \mathsf{mrgn}$	$\in_{\mathsf{mr}} : \mathsf{addr} \times \mathsf{level_K} \times \mathsf{mrgn}$ $\subseteq_{\mathsf{mr}} : \mathsf{mrgn} \times \mathsf{mrgn}$ $\#_{\mathsf{mr}} : \mathsf{mrgn} \times \mathsf{mrgn}$
Σ_{bridge}	mem addr set path	$\mathit{path2set} : \mathsf{path} \to \mathsf{set}$ $\mathit{addr2set}_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{level_K} \to \mathsf{set}$ $\mathit{getp}_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \to \mathsf{path}$ $\mathit{fstlock}_{\mathsf{K}} : \mathsf{mem} \times \mathsf{path} \times \mathsf{level_K} \to \mathsf{addr}$	$\mathit{ordList} : \mathsf{mem} \times \mathsf{path}$

Fig. 4. The signature of the $\mathsf{TSL_K}$ theory

Interpret. of sorts: addr, elem, thid, $\mathsf{level_K}$, ord, cell, mem, path, set, setth and mrgn
Each sort σ in $\Sigma_{\mathsf{TSL_K}}$ is mapped to a non-empty set $\mathcal{A}_\sigma$ such that: (a) $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}_{\mathsf{elem}}$ are discrete sets (b) $\mathcal{A}_{\mathsf{thid}}$ is a discrete set containing $\oslash$ (c) $\mathcal{A}_{\mathsf{level_K}}$ is the finite collection $0,\ldots,\mathsf{K}$-1 (d) $\mathcal{A}_{\mathsf{ord}}$ is a total ordered set (e) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{ord}} \times \mathcal{A}_{\mathsf{addr}}^{\mathsf{K}} \times \mathcal{A}_{\mathsf{thid}}^{\mathsf{K}}$ (f) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$ (g) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$ (h) $\mathcal{A}_{\mathsf{set}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$ (i) $\mathcal{A}_{\mathsf{setth}}$ is the power-set of $\mathcal{A}_{\mathsf{thid}}$ (j) $\mathcal{A}_{\mathsf{mrgn}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{level_K}}$

Signature	Interpretation
Σ_{ord}	$x \preceq^{\mathcal{A}} y \wedge y \preceq^{\mathcal{A}} x \to x = y$ $\quad$ $x \preceq^{\mathcal{A}} y \vee y \preceq^{\mathcal{A}} x$ $\quad$ for any $x, y, z \in \mathcal{A}_{\mathsf{ord}}$ $x \preceq^{\mathcal{A}} y \wedge y \preceq^{\mathcal{A}} z \to x \preceq^{\mathcal{A}} z$ $\quad$ $-\infty^{\mathcal{A}} \preceq^{\mathcal{A}} x \wedge x \preceq^{\mathcal{A}} +\infty^{\mathcal{A}}$
Σ_{cell}	– $mkcell^{\mathcal{A}}(e, k, \overrightarrow{a}, \overrightarrow{t}) = \langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle$ $\quad$ – $error^{\mathcal{A}}.next^{\mathcal{A}} = null^{\mathcal{A}}$ – $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.data^{\mathcal{A}} = e$ $\quad$ – $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.key^{\mathcal{A}} = k$ – $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.next^{\mathcal{A}}[j] = a_j$ $\quad$ – $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.lockid^{\mathcal{A}}[j] = t_j$ – $\langle e, k, \overrightarrow{a}, \ldots t_{j-1}, t_j, t_{j+1} \ldots \rangle.lock^{\mathcal{A}}[j](t') = \langle e, k, \overrightarrow{a}, \ldots t_{j-1}, t', t_{j+1} \ldots \rangle$ – $\langle e, k, \overrightarrow{a}, \ldots t_{j-1}, t_j, t_{j+1} \ldots \rangle.unlock^{\mathcal{A}}[j] = \langle e, k, \overrightarrow{a}, \ldots t_{j-1}, \oslash, t_{j+1} \ldots \rangle$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $k \in \mathcal{A}_{\mathsf{ord}}$, $t_0, \ldots, t_j, t_{j+1}, t_{j-1}, t' \in \mathcal{A}_{\mathsf{thid}}$, $\overrightarrow{a} \in \mathcal{A}_{\mathsf{addr}}^{\mathsf{K}}$, $\overrightarrow{t} \in \mathcal{A}_{\mathsf{thid}}^{\mathsf{K}}$ and $j \in \mathcal{A}_{\mathsf{level_K}}$
Σ_{mem}	$m[a]^{\mathcal{A}} = m(a)$ $\quad$ $upd^{\mathcal{A}}(m, a, c) = m_{a \mapsto c}$ $\quad$ $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$
Σ_{reach}	– $\epsilon^{\mathcal{A}}$ is the empty sequence – $[i]^{\mathcal{A}}$ is the sequence containing $i \in \mathcal{A}_{\mathsf{addr}}$ as the only element – $([i_1 \mathinner{..} i_n], [j_1 \mathinner{..} j_m], [i_1 \mathinner{..} i_n, j_1 \mathinner{..} j_m]) \in append^{\mathcal{A}}$ iff $i_k \neq j_l$. – $(m, a_{init}, a_{end}, l, p) \in reach_{\mathsf{K}}{}^{\mathcal{A}}$ iff $a_{init} = a_{end}$ and $p = \epsilon$, or there exist addresses $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ such that: (a) $p = [a_1 \mathinner{..} a_n]$ $\quad$ (c) $m(a_r).next^{\mathcal{A}}[l] = a_{r+1}$, for $r < n$ (b) $a_1 = a_{init}$ $\quad$ (d) $m(a_n).next^{\mathcal{A}}[l] = a_{end}$
Σ_{mrgn}	– $\mathbf{emp}_{\mathsf{mr}}^{\mathcal{A}} = \emptyset$ $\quad$ – $r \cup_{\mathsf{mr}}^{\mathcal{A}} s = r \cup s$ $\quad$ – $(a, j) \in_{\mathsf{mr}}^{\mathcal{A}} r \leftrightarrow (a, j) \in r$ – $\langle a, j \rangle_{\mathsf{mr}}^{\mathcal{A}} = \{(a, j)\}$ $\quad$ – $r \cap_{\mathsf{mr}}^{\mathcal{A}} s = r \cap s$ $\quad$ – $r \subseteq_{\mathsf{mr}}^{\mathcal{A}} s \leftrightarrow r \subseteq s$ – $r -_{\mathsf{mr}}^{\mathcal{A}} s = r \setminus s$ $\quad$ – $r \#_{\mathsf{mr}}^{\mathcal{A}} s \leftrightarrow r \cap_{\mathsf{mr}}^{\mathcal{A}} s = \mathbf{emp}_{\mathsf{mr}}^{\mathcal{A}}$ for each $a \in \mathcal{A}_{\mathsf{addr}}$, $j \in \mathcal{A}_{\mathsf{level_K}}$ and $r, s \in \mathcal{A}_{\mathsf{mrgn}}$
Σ_{bridge}	– $path2set^{\mathcal{A}}(p) = \{a_1, \ldots, a_n\}$ for $p = [a_1, \ldots, a_n] \in \mathcal{A}_{\mathsf{path}}$ – $addr2set_{\mathsf{K}}{}^{\mathcal{A}}(m, a, l) = \left\{a' \mid \exists p \in \mathcal{A}_{\mathsf{path}} \ .\ (m, a, a', l, p) \in reach_{\mathsf{K}}\right\}$ – $getp_{\mathsf{K}}{}^{\mathcal{A}}(m, a_{init}, a_{end}, l) = \begin{cases} p & \text{if } (m, a_{init}, a_{end}, l, p) \in reach_{\mathsf{K}}{}^{\mathcal{A}} \\ \epsilon & \text{otherwise} \end{cases}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$, $l \in \mathcal{A}_{\mathsf{level_K}}$ and $a_{init}, a_{end} \in \mathcal{A}_{\mathsf{addr}}$ – $fstlock^{\mathcal{A}}(m, [a_1 \mathinner{..} a_n], l) = \begin{cases} a_k & \text{if there is } k \leq n \text{ such that} \\ & \text{for all } j < k, m[a_j].lockid[l] = \oslash \\ & \text{and } m[a_k].lockid[l] \neq \oslash \\ null & \text{otherwise} \end{cases}$ – $ordList^{\mathcal{A}}(m, p)$ iff $p = \epsilon$ or $p = [a]$ or $p = [a_1 \mathinner{..} a_n]$ with $n \geq 2$ and $m(a_i).key^{\mathcal{A}} \preceq m(a_{i+1}).key^{\mathcal{A}}$ for all $1 \leq i < n$, for any $m \in \mathcal{A}_{\mathsf{mem}}$

Fig. 5. Characterization of a $\mathsf{TSL_K}$-interpretation $\mathcal{A}$

4 Decidability of $\mathsf{TSL_K}$

We show that $\mathsf{TSL_K}$ is decidable by proving that it enjoys the finite model property with respect to its sorts, and exhibiting upper bounds for the sizes of the domains of a small interpretation of a satisfiable formula.

Definition 1 (Finite Model Property). *Let Σ be a signature, $S_0 \subseteq S$ be a set of sorts, and T be a Σ-theory. T has the finite model property with respect to S_0 if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ such that for each sort $\sigma \in S_0$, $\mathcal{A}_\sigma$ is finite.*

The fact that $\mathsf{TSL_K}$ has the finite model property with respect to domains elem, addr, ord, $\mathsf{level_K}$ and thid, implies that $\mathsf{TSL_K}$ is decidable by enumerating all possible $\Sigma_{\mathsf{TSL_K}}$-structures up to a certain cardinality. We now define the set of normalized $\mathsf{TSL_K}$-literals.

Definition 2 ($\mathsf{TSL_K}$-normalized literals). *A $\mathsf{TSL_K}$-literal is normalized if it is a flat literal of the form:*

$e_1 \neq e_2$	$a_1 \neq a_2$	$l_1 \neq l_2$
$a = null$	$c = error$	$c = rd(m, a)$
$k_1 \neq k_2$	$k_1 \preceq k_2$	$m_2 = upd(m_1, a, c)$
$c = mkcell(e, k, a_0, \ldots, a_{K-1}, t_0, \ldots, t_{K-1})$		
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$g = \{t\}_T$	$g_1 = g_2 \cup_T g_3$	$g_1 = g_2 \setminus_T g_3$
$r = \langle a, l\rangle_{\mathsf{mr}}$	$r_1 = r_2 \cup_{\mathsf{mr}} r_3$	$r_1 = r_2 -_{\mathsf{mr}} r_3$
$p_1 \neq p_2$	$p = [a]$	$p_1 = rev(p_2)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$
$s = addr2set_K(m, a, l)$	$p = getp_K(m, a_1, a_2, l)$	
$t_1 \neq t_2$	$a = fstlock\,(m, p, l)$	$ordList(m, p)$

where e, e_1 and e_2 are elem*-variables; a, a_0, a_1, $a_2, \ldots, a_{K-1}$ are* addr*-variables; c is a* cell*-variable; m, m_1 and m_2 are* mem*-variables; p, p_1, p_2 and p_3 are* path*-variables; s, s_1, s_2 and s_3 are* set*-variables; g, g_1, g_2 and g_3 are* setth*-variables; r, r_1, r_2 and r_3 are* mrgn*-variables; k, k_1 and k_2 are* ord*-variables; l, l_1 and l_2 are* $\mathsf{level_K}$*-variables and t, t_0, t_1, $t_2, \ldots, t_{K-1}$ are* thid*-variables.*

Lemma 1. *Every $\mathsf{TSL_K}$-formula is equivalent to a collection of conjunctions of normalized $\mathsf{TSL_K}$-literals.*

Proof (sketch). First, transform a formula in disjunctive normal form. Then each conjunct can be normalized introducing auxiliary fresh variables when necessary.

The phase of normalizing a formula is commonly known [15] as the "variable abstraction phase". Note that normalized literals belong to just one theory.

Consider an arbitrary $\mathsf{TSL_K}$-interpretation $\mathcal{A}$ satisfying a conjunction of normalized $\mathsf{TSL_K}$-literals Γ. We show that if $\mathcal{A}$ consists of domains $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$, $\mathcal{A}_{\mathsf{thid}}$, $\mathcal{A}_{\mathsf{level_K}}$ and $\mathcal{A}_{\mathsf{ord}}$ then there are finite sets $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$, $\mathcal{B}_{\mathsf{thid}}$, $\mathcal{B}_{\mathsf{level_K}}$ and $\mathcal{B}_{\mathsf{ord}}$ with bounded cardinalities, where the finite bound on the sizes can be computed from Γ. Such sets can in turn be used to obtain a finite interpretation $\mathcal{B}$ satisfying Γ, since all the other sorts are bounded by the sizes of these sets.

Lemma 2 (Finite Model Property). *Let Γ be a conjunction of normalized $\mathsf{TSL_K}$-literals. Let $\overline{e} = |V_{\mathsf{elem}}(\Gamma)|$, $\overline{a} = |V_{\mathsf{addr}}(\Gamma)|$, $\overline{m} = |V_{\mathsf{mem}}(\Gamma)|$, $\overline{p} = |V_{\mathsf{path}}(\Gamma)|$, $\overline{t} = |V_{\mathsf{thid}}(\Gamma)|$ and $\overline{o} = |V_{\mathsf{ord}}(\Gamma)|$. Then the following are equivalent:*
1. *Γ is $\mathsf{TSL_K}$-satisfiable;*
2. *Γ is true in a $\mathsf{TSL_K}$ interpretation $\mathcal{B}$ such that*

$$
\begin{aligned}
&|\mathcal{B}_{\mathsf{addr}}| \leq \overline{a} + 1 + \overline{m}\,\overline{a}\,\mathsf{K} + \overline{p}^2 + \overline{p}^3 + (\mathsf{K}+2)\overline{m}\,\overline{p} \qquad && |\mathcal{B}_{\mathsf{elem}}| \leq \overline{e} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}| \\
&|\mathcal{B}_{\mathsf{thid}}| \leq \overline{t} + \mathsf{K}\,\overline{m}\,|\mathcal{B}_{\mathsf{addr}}| + 1 && |\mathcal{B}_{\mathsf{ord}}| \leq \overline{o} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}| \\
&|\mathcal{B}_{\mathsf{level_K}}| \leq \mathsf{K}
\end{aligned}
$$

Proof. $(2 \to 1)$ is immediate. $(1 \to 2)$ is proved on a case analysis over the set of normalized literals of $\mathsf{TSL_K}$. □

4.1 A combination-based decision procedure for $\mathsf{TSL_K}$

Lemma 2 enables a brute force method to automatically check whether a set of normalized $\mathsf{TSL_K}$-literals is satisfiable. However, such a method is not efficient in practice. We describe now how to obtain a more efficient decision procedure for $\mathsf{TSL_K}$ applying a many-sorted variant [22] of the Nelson-Oppen combination method [12], by combining the decision procedures for the underlying theories. This combination method requires that the theories fulfill some conditions. First, each theory must have a decision procedure. Second, two theories can only share sorts (but not functions or predicates). Third, when two theories are combined, either both theories are stable infinite or one of them is polite with respect to the underlying sorts that it shares with the other. The stable infinite condition for a theory establishes that if a formula has a model then it has a model with infinite cardinality. In our case, some theories are not stable infinite. For example, $T_{\mathsf{level_K}}$ is not stably infinite, T_{ord}, and T_{thid} need not be stable infinite in same instances. The observation that the condition of stable infinity may be cumbersome in the combination of theories for data structures was already made in [16] where they suggest the condition of *politeness*:

Definition 3 (Politeness). *T is polite with respect to sorts $S : \{\sigma_1 \ldots \sigma_n\}$ whenever:*

(1) Let φ be a satisfiable formula in theory T, $\mathcal{A}$ be one model of φ and let $|\mathcal{A}_{\sigma_1}|, \ldots, |\mathcal{A}_{\sigma_n}|$ be the cardinalities of the domains of $\mathcal{A}$ for sorts in S. For every tuple of larger cardinalities $k_1 \geq |\mathcal{A}_{\sigma_1}|, \ldots, k_n \geq |\mathcal{A}_{\sigma_n}|$, there is a model $\mathcal{B}$ of φ with $|\mathcal{B}_{\sigma_i}| = k_i$.

(2) There is a computable function that for every formula φ returns an equivalent formula $(\exists \overline{v})\psi$ (where $\overline{v} = V_\psi \setminus V_\varphi$) such that, if ψ is satisfiable, then there is an interpretation $\mathcal{A}$ with $\mathcal{A}_\sigma = [V_\sigma(\psi)]^{\mathcal{A}}$ for each sort σ.

Condition *(1)* is called *smoothness*, and guarantees that interpretations can be enlarged as needed. Condition *(2)* is called *finite witnessability*, and gives a procedure to produce a model in which every element is represented by a variable.

The Finite Model Property, Lemma 2 above, guarantees that every sub-theory of $\mathsf{TSL_K}$ is finite witnessable since one can add as many fresh variables as the bound for the corresponding sort in the lemma. The smoothness property can be shown for:

$$T_{\mathsf{cell}} \oplus T_{\mathsf{mem}} \oplus T_{\mathsf{path}} \oplus T_{\mathsf{set}} \oplus T_{\mathsf{setth}} \oplus T_{\mathsf{mrgn}}$$

with respect to sorts addr, $\mathsf{level_K}$, elem, ord and thid. Moreover, these theories can be combined because all of them are stably infinite. The following can also be combined: $T_{\mathsf{level_K}} \oplus T_{\mathsf{ord}} \oplus T_{\mathsf{thid}}$ because they do not share any sorts, so combination is trivial. The many-sorted Nelson-Oppen method allows to combine the first collection of theories with the second. Regarding the decision procedures for each individual theory, $T_{\mathsf{level_K}}$ is trivial since it is just a finite set of naturals with order. For T_{ord} we can adapt a decision procedure for dense orders as the reals [21], or other appropriate theory. For T_{cell} we can use a decision procedure for recursive data structures [13]. T_{mem} is the theory of arrays [1]. T_{set}, T_{setth} and T_{mrgn} are theories of (finite) sets for which there are many decision procedures [25, 8]. The remaining theories are T_{reach} and T_{bridge}. Following the approaches in [16, 18] we extend a decision procedure for the theory T_{path} of finite sequences of (non-repeated) addresses with the auxiliary functions and predicates shown in Fig. 6, and combine this theory to obtain:

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \oplus T_{\mathsf{ord}} \oplus T_{\mathsf{thid}} \oplus T_{\mathsf{level_K}} \oplus T_{\mathsf{cell}} \oplus T_{\mathsf{mem}} \oplus T_{\mathsf{path}} \oplus T_{\mathsf{set}} \oplus T_{\mathsf{setth}} \oplus T_{\mathsf{mrgn}}$$

Using T_{path} all symbols in T_{reach} can be easily defined. The theory of finite sequences of addresses is defined by $T_{\mathsf{fseq}} = (\Sigma_{\mathsf{fseq}}, \mathsf{TGen})$, where $\Sigma_{\mathsf{fseq}} = \big(\{\mathsf{addr}, \mathsf{fseq}\}, \{nil : \mathsf{fseq}, cons : \mathsf{addr} \times \mathsf{fseq} \to \mathsf{fseq}, hd : \mathsf{fseq} \to \mathsf{addr}, tl : \mathsf{fseq} \to \mathsf{fseq}\}, \emptyset\big)$ and TGen as the class of term-generated structures that satisfy the axioms of distinctness, uniqueness and generation of sequences using constructors, as well as acyclicity (see, for example [4]). Let Σ_{path} be Σ_{fseq} extended with the symbols of Fig. 6 and let $PATH$ be the set of axioms of T_{fseq} including the ones in Fig. 6. Then, we can formally define $T_{\mathsf{path}} = (\Sigma_{\mathsf{path}}, \mathsf{ETGen})$ where ETGen is $\big\{\mathcal{A}^{\Sigma_{\mathsf{path}}} | \mathcal{A}^{\Sigma_{\mathsf{path}}} \models PATH \text{ and } \mathcal{A}^{\Sigma_{\mathsf{fseq}}} \in \mathsf{TGen}\big\}$. Next, we extend T_{SLKBase} with definitions for translating all missing functions and predicates from Σ_{reach} and Σ_{bridge} appearing in normalized $\mathsf{TSL_K}$-literals by definitions from T_{SLKBase}. Let GAP be the set of axioms that define ϵ, $[\_]$, $append$, $reach_{\mathsf{K}}$, $path2set$, $getp_{\mathsf{K}}$, $fstlock$ and $ordList$. For instance: $ispath\,(p) \wedge ordPath\,(m,p) \;\leftrightarrow\; ordList\,(m,p)$ We now define $\widehat{\mathsf{TSL_K}} = (\Sigma_{\widehat{\mathsf{TSL_K}}}, \widehat{\mathsf{ETGen}})$ where $\Sigma_{\widehat{\mathsf{TSL_K}}}$ is $\Sigma_{T_{\mathsf{SLKBase}}} \cup \{\; append$, $reach_{\mathsf{K}}$, $path2set$, $getp_{\mathsf{K}}$, $fstlock$, $ordList\; \}$ and $\widehat{\mathsf{ETGen}} := \big\{\mathcal{A}^{\Sigma_{\widehat{\mathsf{TSL_K}}}} | \mathcal{A}^{\Sigma_{\widehat{\mathsf{TSL_K}}}} \models GAP \text{ and } \mathcal{A}^{\Sigma_{T_{\mathsf{SLKBase}}}} \in \mathsf{ETGen}\big\}$.

Using the definitions of GAP it is easy to prove that if Γ is a set of normalized $\mathsf{TSL_K}$-literals, then Γ is $\mathsf{TSL_K}$-satisfiable iff Γ is $\widehat{\mathsf{TSL_K}}$-satisfiable. Therefore, $\widehat{\mathsf{TSL_K}}$ can be used in place of $\mathsf{TSL_K}$ for satisfiability checking. The reduction from $\widehat{\mathsf{TSL_K}}$ into T_{SLKBase} is performed in two steps. First, by the finite model theorem (Lemma 2), it is always possible to calculate an upper bound in the number of elements of sort addr, elem, thid, ord and level in a model (if there is one model), based on the input formula. Therefore, one can introduce one variable

$app : \mathsf{fseq} \times \mathsf{fseq} \to \mathsf{fseq}$
$app(nil, l) = l \qquad app(cons(a, l), l') = cons(a, app(l, l'))$

$fseq2set : \mathsf{fseq} \to \mathsf{set}$
$fseq2set(nil) = \emptyset \qquad fseq2set(cons(a, l)) = \{a\} \cup fseq2set(l)$

$ispath : \mathsf{fseq}$
$ispath(nil) \qquad ispath(cons(a, nil)) \qquad \{a\} \nsubseteq fseq2set(l) \wedge ispath(l) \to ispath(cons(a, l))$

$last : \mathsf{fseq} \to \mathsf{addr}$
$last(cons(a, nil)) = a \qquad l \neq nil \to last(cons(a, l)) = last(l)$

$isreach_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level}_{\mathsf{K}}$
$isreach_{\mathsf{K}}(m, a, a, l) \qquad m[a].next[l] = a' \wedge isreach_{\mathsf{K}}(m, a', b, l) \to isreach_{\mathsf{K}}(m, a, b, l)$

$isreachp_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level}_{\mathsf{K}} \times \mathsf{fseq}$
$isreachp_{\mathsf{K}}(m, a, a, l, nil)$ $m[a].next[l] = a' \wedge isreachp(m, a', b, l, p) \to isreachp(m, a, b, l, cons(a, p))$

$fstmark : \mathsf{mem} \times \mathsf{fseq} \times \mathsf{level}_{\mathsf{K}} \times \mathsf{addr}$
$fstmark(m, nil, l, null)$ $p \neq nil \wedge p = cons(a, q) \wedge m[a].lockid[l] \neq \oslash \to fstmark(m, p, l, a)$ $p \neq nil \wedge p = cons(a, q) \wedge m[a].lockid[l] = \oslash \wedge fstmark(m, q, l, b) \to fstmark(m, p, l, b)$

$ordPath : \mathsf{mem} \times \mathsf{fseq}$
$ordPath(h, nil)$ $\begin{pmatrix} h[a].next[0] = a' \wedge h[a].key \preceq h[a'].key \;\wedge \\ p = cons(a, q) \;\wedge\; ordPath(h, q) \end{pmatrix} \to ordPath(h, p)$

Fig. 6. Functions, predicates and axioms of T_{path}

per element of each of these sorts and unfold all definitions in $PATH$ and GAP, by symbolic expansion, leading to terms in Σ_{fseq}, and thus, in T_{SLKBase}. This way, it is always possible to reduce a $\widehat{\mathsf{TSL_K}}$-satisfiability problem of normalized literals into a T_{SLKBase}-satisfiability problem. Hence, using a decision procedure for T_{SLKBase} we obtain a decision procedure for $\widehat{\mathsf{TSL_K}}$, and thus, for $\mathsf{TSL_K}$. Notice, for instance, that the predicate $subPath : \mathsf{path} \times \mathsf{path}$ for ordered lists can be defined using only $path2set$ as: $subPath(p_1, p_2) \mathrel{\hat{=}} path2set(p_1) \subseteq path2set(p_2)$

For space reasons, we do not provide complete specification and proofs of the temporal properties. However, in [18] is detailed an example of a termination proof over concurrent lists, which easily carries over to skiplists. For illustration purposes, we now show the full verification condition for the verification of the safety property $\square\big(SkipList_4(h, sl)\big)$ when executing transition 36 of program *insert* by a thread with id t, from Section 2. For clarity, we again use *prev* as a short for $upd^{[t]}[i^{[t]}]$, and we use the auxiliary predicate $setnext(c, d, i, x)$ that makes the cell d identical to c except that $c.next[i] = x$.

$$setnext(c, d, i, x) \mathrel{\hat{=}} \begin{pmatrix} d.data = c.data \wedge d.key = c.key \wedge d.lock[j] = c.lock[j] \wedge \\ (i \neq j) \to d.next[j] = c.next[j] \wedge d.next[i] = x \end{pmatrix}$$

The VC is $(SkipList_4(h, sl) \wedge \varphi \rightarrow SkipList_4(h', sl'))$ where φ is:

$$\left(\begin{array}{rr} x^{[t]}.key = newval & \wedge \\ prev.key < newval & \wedge \\ x^{[t]}.next[i^{[t]}].key > newval & \wedge \\ prev.next[i^{[t]}] = x^{[t]}.next[i^{[t]}] & \wedge \\ (x^{[t]}, i^{[t]}) \notin sl.r \wedge 0 \leq i^{[t]} \leq 3 & \end{array}\right) \wedge \left(\begin{array}{cr} at_{36}[t] \wedge at'_{37}[t] & \wedge \\ prev'.next[i^{[t]}] = x^{[t]} & \wedge \\ setnext(h[prev], newcell, i^{[t]}, x^{[t]}) & \wedge \\ h' = upd(h, prev, newcell) & \wedge \\ sl = sl' \wedge x'^{[t]} = x^{[t]} & \wedge \end{array}\right)$$

5 Conclusion and Future Work

In this paper we have presented $\mathsf{TSL_K}$, a theory of skiplists of height at most K, useful for automatically prove the VCs generated during the verification of concurrent skiplist implementations. $\mathsf{TSL_K}$ is capable of reasoning about memory, cells, pointers, masked regions and reachability, enabling ordered lists and sublists, allowing the description of the skiplist property, and the representation of memory modifications introduced by the execution of program statements.

We showed that $\mathsf{TSL_K}$ is decidable by proving its finite model property, and exhibiting the minimal cardinality of a model if one such model exists. Moreover, we showed how to reduce the satisfiability problem of quantifier-free $\mathsf{TSL_K}$ formulas to a combination of theories using the many-sorted version of Nelson-Oppen, allowing the use of well studied decision procedures. The complexity of the decision problem for $\mathsf{TSL_K}$ is easily shown to be NP-complete since it properly extends TLL [16].

Current work includes the translation of formulas from T_{ord}, $T_{\mathsf{level_K}}$, T_{set}, T_{setth} and T_{mrgn} into BAPA [8]. In BAPA, arithmetic, sets and cardinality aids in the definition of skiplists properties. Paths can be represented as finite sequences of addresses. We are studying how to replace the recursive functions from T_{reach} and Σ_{bridge} by canonical set and list abstractions [20], which would lead to a more efficient decision procedure, essentially encoding full $\mathsf{TSL_K}$ formulas into BAPA. The family of theories presented in the paper is limited to skiplists of a fixed maximum height. Typical skiplist implementations fix a maximum number of levels and this can be handled with $\mathsf{TSL_K}$. Inserting more than than 2^{levels} elements into a skiplist may slow-down the search of a skiplist implementation but this issue affects performance and not correctness, which is the goal pursued in this paper. We are studying techniques to describe skiplists of arbitrary many levels. A promising approach consists of equipping the theory with a primitive predicate denoting that the skiplist property holds above and below a given level. Then the reasoning is restricted to the single level being modified. This approach, however, is still work in progress.

Furthermore, we are working on a direct implementation of our decision procedure, as well as its integration into existing solvers. Future work also includes the temporal verification of sequential and concurrent skiplists implementations, including one at the `java.concurrent` standard library. This can be accomplished by the design of verification diagrams that use the decision procedure presented in this paper.

References

1. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Information and Computation 183(2), 140–164 (2003)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Proc. of ECOOP'08. pp. 387–411. Springer (2008)
3. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: CONCUR'09. pp. 178–195 (2009)
4. Bradley, A.R., Manna, Z.: The Calculus of Computation. Springer-Verlag (2007)
5. Browne, A., Manna, Z., Sipma, H.B.: Generalized verification diagrams. In: Proc. of FSTTCS'95. LNCS, vol. 1206, pp. 484–498. Springer (1995)
6. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgran-Kaufmann (2008)
7. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Proc. of ESOP'08. LNCS, vol. 4960, pp. 353–367. Springer (2008)
8. Kuncak, V., Nguyen, H.H., Rinard, M.C.: An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In: CADE'05. pp. 260–277 (2005)
9. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: Proc. of POPL'08. pp. 171–182. ACM (2008)
10. Leino, K.R.M.: Data groups: Specifying the modication of extended state. In: OOP-SLA'98. pp. 144–153. ACM (1998)
11. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer (1995)
12. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
13. Oppen, D.C.: Reasoning about recursively defined data structures. J. ACM 27(3), 403–411 (1980)
14. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)
15. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: FROCOS'05. pp. 48–64 (2005)
16. Ranise, S., Zarba, C.G.: A theory of singly-linked lists and its extensible decision procedure. In: Proc. of SEFM 2006. IEEE CS Press (2006)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of LICS'02. pp. 55–74. IEEE CS Press (2002)
18. Sánchez, A., Sánchez, C.: Decision procedures for the temporal verification of concurrent lists. In: Proc. of ICFEM'10. LNCS, vol. 6447, pp. 74–89. Springer (2010)
19. Sipma, H.B.: Diagram-Based Verification of Discrete, Real-Time and Hybrid Systems. Ph.D. thesis, Stanford University (1999)
20. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: Proc. of POPL'10. pp. 199–210. ACM (2010)
21. Tarski, A.: A decision method for elementary algebra and geometry. University of California Press (1951)
22. Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: JELIA'04. LNCS, vol. 3229, pp. 641–653. Springer (2004)
23. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, Unversity of Cambridge (2007)
24. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: FOSSACS'06. pp. 94–110 (2006)
25. Zarba, C.G.: Combining sets with elements. In: Verification: Theory and Practice. LNCS, vol. 2772, pp. 762–782. Springer (2003)

A Small Model Property

A.1 Normalized Literals

We show that $\mathsf{TSL_K}$ also has the finite model property with respect to domains elem, addr, thid, ord and level. Hence, $\mathsf{TSL_K}$ is decidable because one can enumerate $\Sigma_{\mathsf{TSL_K}}$-structures up to a certain cardinality. To prove this result, we first define the set of normalized $\mathsf{TSL_K}$-literals.

Definition 2 ($\mathsf{TSL_K}$-normalized literals). *A $\mathsf{TSL_K}$-literal is normalized if it is a flat literal of the form:*

$$
\begin{array}{lll}
e_1 \neq e_2 & a_1 \neq a_2 & l_1 \neq l_2 \\
a = null & c = error & \\
k_1 \neq k_2 & k_1 \preceq k_2 & \\
\multicolumn{3}{l}{c = mkcell(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})} \\
c = rd(m, a) & m_2 = upd(m_1, a, c) & \\
s = \{a\} & s_1 = s_2 \cup s_3 & s_1 = s_2 \setminus s_3 \\
g = \{t\}_T & g_1 = g_2 \cup_T g_3 & g_1 = g_2 \setminus_T g_3 \\
r = \langle a, l \rangle_{\mathsf{mr}} & r_1 = r_2 \cup_{\mathsf{mr}} r_3 & r_1 = r_2 -_{\mathsf{mr}} r_3 \\
p_1 \neq p_2 & p = [a] & p_1 = rev(p_2) \\
s = path2set(p) & append(p_1, p_2, p_3) & \neg append(p_1, p_2, p_3) \\
s = addr2set_{\mathsf{K}}(m, a, l) & p = getp_{\mathsf{K}}(m, a_1, a_2, l) & \\
t_1 \neq t_2 & a = fstlock\,(m, p, l) & ordList(m, p)
\end{array}
$$

where e, e_1 and e_2 are elem-variables; a, a_1, $a_2, \ldots, a_{\mathsf{K}-1}$ are addr-variables; c is a cell-variable; m, m_1 and m_2 are mem-variables; p, p_1, p_2 and p_3 are path-variables; s, s_1, s_2 and s_3 are set-variables; g, g_1, g_2 and g_3 are setth-variables; r, r_1, r_2 and r_3 are mrgn-variables; k, k_1 and k_2 are ord-variables; l, l_1 and l_2 are $\mathsf{level_K}$-variables and t, t_1, $t_2, \ldots, t_{\mathsf{K}-1}$ are thid-variables.

The remaining literals can be rewritten from the normalized ones using the following equivalences:

$$
\begin{array}{ll}
e = c.data & \leftrightarrow (\exists_{\mathsf{ord}} k\ \exists_{\mathsf{addr}} a_0, \ldots, a_{\mathsf{K}-1}\ \exists_{\mathsf{thid}} t_0, \ldots, t_{\mathsf{K}-1}) \\
& \qquad [c = mkcell\,(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})] \\
k = c.key & \leftrightarrow (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{addr}} a_0, \ldots, a_{\mathsf{K}-1}\ \exists_{\mathsf{thid}} t_0, \ldots, t_{\mathsf{K}-1}) \\
& \qquad [c = mkcell\,(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})] \\
a = c.next[l] & \leftrightarrow (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{ord}} k\ \exists_{\mathsf{addr}} a_0, \ldots, a_{l-1}, a_{l+1}, \ldots, a_{\mathsf{K}-1}\ \exists_{\mathsf{thid}} t_0, \ldots, t_{\mathsf{K}-1}) \\
& \qquad [c = mkcell\,(e, k, a_0, \ldots, a_{l-1}, a, a_{l+1}, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})] \\
t = c.lockid[l] & \leftrightarrow (\exists_{\mathsf{elem}} e\ \exists_{\mathsf{ord}} k\ \exists_{\mathsf{addr}} a_0, \ldots, a_{\mathsf{K}-1}\ \exists_{\mathsf{thid}} t_0, \ldots, t_{l-1}, t_{l+1}, \ldots, t_{\mathsf{K}-1}) \\
& \qquad [c = mkcell\,(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{l-1}, t, t_{l+1}, \ldots, t_{\mathsf{K}-1})]
\end{array}
$$

$$
\begin{aligned}
c_1 = c_2.\mathit{lock}\,(l,t) \;\leftrightarrow\;& c_2.\mathit{data} = c_1.\mathit{data} \wedge c_2.\mathit{key} = c_1.\mathit{key} \wedge \\
& \quad c_2.\mathit{next}[0] = c_1.\mathit{next}[0] \wedge \\
& \quad \ldots \\
& \quad c_2.\mathit{next}[\mathsf{K}-1] = c_1.\mathit{next}[\mathsf{K}-1] \wedge \\
& \quad c_2.\mathit{lockid}[0] = c_1.\mathit{lockid}[0] \wedge \\
& \quad \ldots \\
& \quad c_2.\mathit{lockid}[l-1] = c_1.\mathit{lockid}[l-1] \wedge \\
& \quad t = c_1.\mathit{lockid}[l] \wedge \\
& \quad c_2.\mathit{lockid}[l+1] = c_1.\mathit{lockid}[l+1] \wedge \\
& \quad \ldots \\
& \quad c_2.\mathit{lockid}[\mathsf{K}-1] = c_1.\mathit{lockid}[\mathsf{K}-1] \\
c_1 = c_2.\mathit{unlock}\,(l) \;\leftrightarrow\;& c_2.\mathit{data} = c_1.\mathit{data} \wedge c_2.\mathit{key} = c_1.\mathit{key} \wedge \\
& \quad c_2.\mathit{next}[0] = c_1.\mathit{next}[0] \wedge \\
& \quad \ldots \\
& \quad c_2.\mathit{next}[\mathsf{K}-1] = c_1.\mathit{next}[\mathsf{K}-1] \wedge \\
& \quad c_2.\mathit{lockid}[0] = c_1.\mathit{lockid}[0] \wedge \\
& \quad \ldots \\
& \quad c_2.\mathit{lockid}[l-1] = c_1.\mathit{lockid}[l-1] \wedge \\
& \quad \oslash = c_1.\mathit{lockid}[l] \wedge \\
& \quad c_2.\mathit{lockid}[l+1] = c_1.\mathit{lockid}[l+1] \wedge \\
& \quad \ldots \\
& \quad c_2.\mathit{lockid}[\mathsf{K}-1] = c_1.\mathit{lockid}[\mathsf{K}-1] \\
c_1 \neq_{\mathsf{cell}} c_2 \;\leftrightarrow\;& c_1.\mathit{data} \neq c_2.\mathit{data} \vee c_1.\mathit{key} \neq c_2.\mathit{key} \vee \\
& c_1.\mathit{next}[0] \neq c_2.\mathit{next}[0] \vee \\
& \ldots \\
& c_1.\mathit{next}[\mathsf{K}-1] \neq c_2.\mathit{next}[\mathsf{K}-1] \vee \\
& c_1.\mathit{lockid}[0] \neq c_2.\mathit{next}[0] \vee \\
& \ldots \\
& c_1.\mathit{lockid}[\mathsf{K}-1] \neq c_2.\mathit{next}[\mathsf{K}-1] \\
m_1 \neq_{\mathsf{mem}} m_2 \;\leftrightarrow\;& (\exists_{\mathsf{addr}} a)\,[\mathit{rd}(m_1,a) \neq \mathit{rd}(m_2,a)] \\
g_1 \neq_{\mathsf{setth}} g_2 \;\leftrightarrow\;& (\exists_{\mathsf{thid}} t)\,[t \in (g_1 \setminus_T g_2) \cup_T (g_2 \setminus_T g_1)] \\
g = \emptyset_T \;\leftrightarrow\;& g = g \setminus_T g \\
g_3 = g_1 \cap_T g_2 \;\leftrightarrow\;& g_3 = (g_1 \cup_T g_2) \setminus_T ((g_1 \setminus_T g_2) \cup_T (g_2 \setminus_T g_1)) \\
t \in_T g \;\leftrightarrow\;& \{t\}_T \subseteq_T g \\
g_1 \subseteq_T g_2 \;\leftrightarrow\;& g_2 = g_1 \cup_T g_2
\end{aligned}
$$

$$
\begin{array}{ll}
r_1 \neq_{\mathsf{setth}} r_2 & \leftrightarrow (\exists_{\mathsf{addr}} a\ \exists_{\mathsf{level_K}} l)\,[(a,l) \in (r_1 -_{\mathsf{mr}} r_2) \cup_{\mathsf{mr}} (r_2 -_{\mathsf{mr}} r_1)] \\
r = \mathbf{emp}_{\mathsf{mr}} & \leftrightarrow r = r -_{\mathsf{mr}} r \\
r_3 = r_1 \cap_{\mathsf{mr}} r_2 & \leftrightarrow r_3 = (r_1 \cup_{\mathsf{mr}} r_2) -_{\mathsf{mr}} ((r_1 -_{\mathsf{mr}} r_2) \cup_{\mathsf{mr}} (r_2 -_{\mathsf{mr}} r_1)) \\
(a,l) \in_{\mathsf{mr}} r & \leftrightarrow \langle a,l \rangle_{\mathsf{mr}} \subseteq_{\mathsf{mr}} r \\
r_1 \subseteq_{\mathsf{mr}} r_2 & \leftrightarrow r_2 = r_1 \cup_{\mathsf{mr}} r_2 \\
r_1 \#_{\mathsf{mr}} r_2 & \leftrightarrow \mathbf{emp}_{\mathsf{mr}} = (r_1 \cup_{\mathsf{mr}} r_2) -_{\mathsf{mr}} ((r_1 -_{\mathsf{mr}} r_2) \cup_{\mathsf{mr}} (r_2 -_{\mathsf{mr}} r_1)) \\
p = \epsilon & \leftrightarrow \mathit{append}(p,p,p) \\
\mathit{reach}_{\mathsf{K}}(m,a_1,a_2,l,p) & \leftrightarrow a_2 \in \mathit{addr2set}_{\mathsf{K}}(m,a_1,l) \wedge p = \mathit{getp}_{\mathsf{K}}(m,a_1,a_2,l)
\end{array}
$$

this means that we can rewrite such literals using:

Flat:	$e = c.\mathit{data}$
Normalized:	$c = \mathit{mkcell}(e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{\mathsf{K}-1})$
Proviso:	$k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$k = c.\mathit{key}$
Normalized:	$c = \mathit{mkcell}(e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{\mathsf{K}-1})$
Proviso:	$e,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$a = c.\mathit{next}[l]$
Normalized:	$c = \mathit{mkcell}(e,k,a_0,\dots,a_{l-1},a,a_{l+1},\dots,a_{\mathsf{K}-1},t_0,\dots,t_{\mathsf{K}-1})$
Proviso:	$e,k,a_0,\dots,a_{l-1},a_{l+1},a_{\mathsf{K}-1},t_0,\dots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$t = c.\mathit{lockid}[l]$
Normalized:	$c = \mathit{mkcell}(e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{l-1},t,t_{l+1},\dots,t_{\mathsf{K}-1})$
Proviso:	$e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{l-1},t_{l+1},\dots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$c_1 = c_2.\mathit{lock}(l,t)$
Normalized:	$c_1 = \mathit{mkcell}(e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{l-1},t,t_{l+1},\dots,t_{\mathsf{K}-1}) \wedge$ $c_2 = \mathit{mkcell}(e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{l-1},\tilde{t},t_{l+1},\dots,t_{\mathsf{K}-1})$
Proviso:	$e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{l-1},\tilde{t},t_{l+1},\dots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$c_1 = c_2.\mathit{unlock}(l)$
Normalized:	$c_1 = \mathit{mkcell}(e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{l-1},\oslash,t_{l+1},\dots,t_{\mathsf{K}-1}) \wedge$ $c_2 = \mathit{mkcell}(e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{l-1},\tilde{t},t_{l+1},\dots,t_{\mathsf{K}-1})$
Proviso:	$e,k,a_0,\dots,a_{\mathsf{K}-1},t_0,\dots,t_{l-1},\tilde{t},t_{l+1},\dots,t_{\mathsf{K}-1}$ are fresh.
Flat:	$c_1 \neq c_2$
Normalized:	$c_1.\mathit{data} \neq c_2.\mathit{data} \vee c_1.\mathit{key} \neq c_2.\mathit{key} \vee$ $c_1.\mathit{next}[0] \neq c_2.\mathit{next}[0] \vee \cdots \vee c_1.\mathit{next}[\mathsf{K}-1] \neq c_2.\mathit{next}[\mathsf{K}-1] \vee$ $c_1.\mathit{lockid}[0] \neq c_2.\mathit{lockid}[0] \vee \cdots \vee c_1.\mathit{lockid}[\mathsf{K}-1] \neq c_2.\mathit{lockid}[\mathsf{K}-1]$
Proviso:	-

Flat:	$m_1 \neq m_2$
Normalized:	$m[a] \neq m[b]$
Proviso:	a is fresh.
Flat:	$s_1 \neq s_2$
Normalized:	$s_{12} = s_1 \setminus s_2 \wedge s_{21} = s_2 \setminus s_1 \wedge s_3 = s_{12} \cup s_{21} \wedge s = s_3 \cup \{a\} \wedge \{a\} \subseteq s$
Proviso:	s_{12}, s_{21}, s_3, s and a are fresh.
Flat:	$s = \emptyset$
Normalized:	$s = s \setminus s$
Proviso:	-
Flat:	$s_3 = s_1 \cap s_2$
Normalized:	$s_{12} = s_1 \setminus s_2 \wedge s_{21} = s_2 \setminus s_1 \wedge s_{u_1} = s_1 \cup s_2 \wedge s_{u_2} = s_{12} \cup s_{21} \wedge$ $s_3 = s_{u_1} \setminus s_{u_2}$
Proviso:	s_{12}, s_{21}, s_{u_1} and s_{u_2} are fresh.
Flat:	$a \in s$
Normalized:	$s = \{a\} \cup s$
Proviso:	-
Flat:	$s_1 \subseteq s_2$
Normalized:	$s_2 = s_1 \cup s_2$
Proviso:	-
Flat:	$g_1 \neq g_2$
Normalized:	$g_{12} = g_1 \setminus_T g_2 \wedge g_{21} = g_2 \setminus_T g_1 \wedge g_3 = g_{12} \cup_T g_{21} \wedge g = g_3 \cup_T \{a\} \wedge$ $\{a\} \subseteq_T g$
Proviso:	g_{12}, g_{21}, g_3, g and a are fresh.
Flat:	$g = \emptyset_T$
Normalized:	$g = g \setminus_T g$
Proviso:	-
Flat:	$g_3 = g_1 \cap_T g_2$
Normalized:	$g_{12} = g_1 \setminus_T g_2 \wedge g_{21} = g_2 \setminus_T g_1 \wedge g_{u_1} = g_1 \cup_T g_2 \wedge g_{u_2} = g_{12} \cup_T g_{21} \wedge$ $g_3 = g_{u_1} \setminus_T g_{u_2}$
Proviso:	g_{12}, g_{21}, g_{u_1} and g_{u_2} are fresh.
Flat:	$t \in_T g$
Normalized:	$g = \{t\} \cup_T g$
Proviso:	-
Flat:	$g_1 \subseteq_T g_2$
Normalized:	$g_2 = g_1 \cup_T g_2$
Proviso:	-
Flat:	$r_1 \neq r_2$
Normalized:	$r_{12} = r_1 -_{\mathsf{mr}} r_2 \wedge r_{21} = r_2 -_{\mathsf{mr}} r_1 \wedge r_3 = r_{12} \cup_{\mathsf{mr}} r_{21} \wedge$ $r = r_3 \cup_{\mathsf{mr}} \{(a, l)\} \wedge \{(a, l)\} \subseteq_{\mathsf{mr}} r$
Proviso:	r_{12}, r_{21}, r_3, r, a and l are fresh.
Flat:	$r = \mathbf{emp}_{\mathsf{mr}}$
Normalized:	$r = r -_{\mathsf{mr}} r$
Proviso:	-
Flat:	$r_3 = r_1 \cap_{\mathsf{mr}} r_2$
Normalized:	$r_{12} = r_1 -_{\mathsf{mr}} r_2 \wedge r_{21} = r_2 -_{\mathsf{mr}} r_1 \wedge r_{u_1} = r_1 \cup_{\mathsf{mr}} r_2 \wedge$ $r_{u_2} = r_{12} \cup_{\mathsf{mr}} r_{21} \wedge r_3 = r_{u_1} -_{\mathsf{mr}} r_{u_2}$
Proviso:	r_{12}, r_{21}, r_{u_1} and r_{u_2} are fresh.

Flat:	$(a, l) \in_{\mathsf{mr}} r$
Normalized:	$r = \{(a, l)\} \cup_{\mathsf{mr}} r$
Proviso:	-
Flat:	$r_1 \subseteq_{\mathsf{mr}} r_2$
Normalized:	$r_2 = r_1 \cup_{\mathsf{mr}} r_2$
Proviso:	-
Flat:	$r_1 \#_{\mathsf{mr}} r_2$
Normalized:	$r_{12} = r_1 -_{\mathsf{mr}} r_2 \wedge r_{21} = r_2 -_{\mathsf{mr}} r_1 \wedge r_{u_1} = r_1 \cup_{\mathsf{mr}} r_2 \wedge$ $r_{u_2} = r_{12} \cup_{\mathsf{mr}} r_{21} \wedge r_3 = r_{u_1} -_{\mathsf{mr}} r_{u_2} \wedge r_3 = r_3 -_{\mathsf{mr}} r_3$
Proviso:	r_{12}, r_{21}, r_{u_1}, r_{u_2} and r_3 are fresh.
Flat:	$p = \epsilon$
Normalized:	$\mathit{append}(p, p, p)$
Proviso:	-
Flat:	$\mathit{reach}_{\mathsf{K}}(m, a_1, a_2, l, p)$
Normalized:	$a_2 \in \mathit{addr2set}_{\mathsf{K}}(m, a_1, l) \wedge p = \mathit{getp}_{\mathsf{K}}(m, a_1, a_2)$
Proviso:	-

A.2 The Small Model Property

Consider an arbitrary $\mathsf{TSL}_{\mathsf{K}}$-interpretation $\mathcal{A}$ satisfying a conjunction of normalized $\mathsf{TSL}_{\mathsf{K}}$-literals Γ. We show that if there are sets $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$, $\mathcal{A}_{\mathsf{thid}}$, $\mathcal{A}_{\mathsf{level_K}}$ and $\mathcal{A}_{\mathsf{ord}}$ then there are finite sets $\mathcal{A}'_{\mathsf{elem}}$, $\mathcal{A}'_{\mathsf{addr}}$, $\mathcal{A}'_{\mathsf{thid}}$, $\mathcal{A}'_{\mathsf{level_K}}$ and $\mathcal{A}'_{\mathsf{ord}}$ with bounded cardinalities (the bound depending on Γ). $\mathcal{A}'_{\mathsf{elem}}$, $\mathcal{A}'_{\mathsf{addr}}$, $\mathcal{A}'_{\mathsf{thid}}$, $\mathcal{A}'_{\mathsf{level_K}}$ and $\mathcal{A}'_{\mathsf{ord}}$ can in turn be used to obtain a finite interpretation $\mathcal{A}'$ satisfying Γ.

Before proving that $\mathsf{TSL}_{\mathsf{K}}$ enjoys of finite model property, we define some auxiliary functions. We start by defining the function $\mathit{first}_{\mathsf{K}}$. Let $\mathcal{B}_{\mathsf{addr}} \subseteq \tilde{\mathcal{B}}_{\mathsf{addr}}$, $m : \tilde{\mathcal{B}}_{\mathsf{addr}} \to \mathcal{B}_{\mathsf{elem}} \times \mathcal{B}_{\mathsf{ord}} \times \tilde{\mathcal{B}}^{\mathsf{K}}_{\mathsf{addr}} \times \mathcal{B}^{\mathsf{K}}_{\mathsf{thid}}$, $a \in \mathcal{B}_{\mathsf{addr}}$ and $l \in \mathcal{B}_{\mathsf{level_K}}$. The function $\mathit{first}_{\mathsf{K}}(m, a, l, \mathcal{B}_{\mathsf{addr}})$ is defined by

$$\mathit{first}_{\mathsf{K}}(m, a, l, \mathcal{B}_{\mathsf{addr}}) = \begin{cases} \mathit{null} & \text{if for all } r \geq 1 \;\; m^r(a).\mathit{next}(l) \notin \mathcal{B}_{\mathsf{addr}} \\ m^s(a).\mathit{next}(l) & \text{if for some } s \geq 1 \;\; m^s(a).\mathit{next}(l) \in \mathcal{B}_{\mathsf{addr}}, \\ & \quad \text{and for all } r < s \;\; m^r(a).\mathit{next}(l) \notin \mathcal{B}_{\mathsf{addr}} \end{cases}$$

where

- $m^1(a).\mathit{next}(l)$ stands for $m(a).\mathit{next}(l)$ and
- $m^{n+1}(a).\mathit{next}(l)$ stands for $m(m^n(a).\mathit{next}(l)).\mathit{next}(l)$ when $n > 0$.

Basically, given the original model $\mathcal{A}$ and a subset of addresses $X \subseteq \mathcal{A}_{\mathsf{addr}}$, function $\mathit{first}_{\mathsf{K}}$ chooses the next address in X that can be reached from a given address following repeatedly the $\mathit{next}(l)$ pointer. It is easy to see, for example, that if $m(a).\mathit{next}(l) \in X$ then $\mathit{first}_{\mathsf{K}}(m, a, l, X) = m(a).\mathit{next}(l)$. We will later filter out unnecessary intermediate nodes and use $\mathit{first}_{\mathsf{K}}$ to bypass properly the removed nodes, preserving the important connectivity properties.

Lemma 3. *Let $\mathcal{B}_{\mathsf{addr}} \subseteq \tilde{\mathcal{B}}_{\mathsf{addr}}$, $m : \tilde{\mathcal{B}}_{\mathsf{addr}} \to \mathcal{B}_{\mathsf{elem}} \times \mathcal{B}_{\mathsf{ord}} \times \tilde{\mathcal{B}}^{K}_{\mathsf{addr}} \times \mathcal{B}^{K}_{\mathsf{thid}}$, $a \in \mathcal{B}_{\mathsf{addr}}$ and $l \in \mathcal{B}_{\mathsf{level_K}}$. If $m(a).\mathit{next}(l) \in \mathcal{B}_{\mathsf{addr}}$, then $\mathit{first}_K(m, a, l, \mathcal{B}_{\mathsf{addr}}) = m(a).\mathit{next}(l)$.*

Proof. Immediate from definition of $first_{\mathsf{K}}$.

Secondly, we define the *compress* function which, given a path p and a set $\mathcal{B}_{\mathsf{addr}}$ of addresses, returns the path obtained from p by removing all the addresses that do not belong to $\mathcal{B}_{\mathsf{addr}}$.

$$compress([i_1, \ldots, i_n], \mathcal{B}_{\mathsf{addr}}) = \begin{cases} \epsilon & \text{if } n = 0 \\ [i_1] \circ compress([i_2, \ldots, i_n], X) & \text{if } n > 0 \text{ and } i_1 \in X \\ compress([i_2, \ldots, i_n], X) & \text{otherwise} \end{cases}$$

Third, the function *fstL* that, given a memory, a path and a level, chooses the first address in a path (at the given level), whose lock is not $\oslash$, returning the address as a singleton set:

$$fstL(m, [i_1, \ldots, i_n], l) = \begin{cases} \emptyset & \text{if } n = 0 \\ \{i_1\} & \text{if } m(i_1).lockid(l) \neq \oslash \\ fstL(m, [i_2, \ldots, i_n], l) & \text{if } m(i_1).lockid(l) = \oslash \end{cases}$$

Fourth, the function *unordered* that given a memory m and a path p, returns a set containing two address that witness the failure to preserve the *key* order of elements in p:

$$unordered(m, [i_1, \ldots, i_n]) = \begin{cases} \emptyset & \text{if } n = 0 \text{ or } n = 1 \\ \{i_1, i_2\} & \text{if } m(i_2).key \preceq m(i_1).key \text{ and} \\ & \quad m(i_2).key \neq m(i_1).key \\ unordered(m, [i_2, \ldots, i_n]) & \text{otherwise} \end{cases}$$

If two such addresses exist, *unordered* returns the first two consecutive addresses whose keys violate the order.

Lemma 4. *Let p be a path such that $p = [a_1, \ldots, a_n]$ with $n \geq 2$ and let m be a memory. If exists a_i, with $1 \leq i < n$, such that $m(a_{i+1}).key \preceq m(a_i).key$ and $m(a_{i+1}).key \neq m[a_i].key$, then $unordered(m, p) \neq \emptyset$*

Proof. By induction. Let's consider $n = 2$ and let $p = [a_1, a_2]$ s.t., $m(a_2).key \preceq m(a_1).key$ and $m(a_2).key \neq m(a_1).key$. Then, by definition of *unordered*, we have that $unordered(m, p) = \{a_1, a_2\} \neq \emptyset$.

Now let's assume that $n > 2$ and let $p = [a_1, \ldots, a_n]$. If $m(a_2).key \preceq m(a_1).key$ and $m(a_2).key \neq m(a_1).key$, then we have that $unordered(m, p) = \{a_1, a_2\} \neq \emptyset$. On the other hand, if $m(a_1).key \preceq m(a_2).key$, we still knows that there is a a_i, with $2 \leq i < n$, s.t., $m(a_{i+1}).key \preceq m(a_i).key$ and $m(a_{i+1}).key \neq m(a_i).key$. Therefore, by induction we have that $unordered(m, [a_2, \ldots, a_n]) \neq \emptyset$ and by definition of *unordered*, $unordered(m, p) = unordered(m, [a_2, \ldots, a_n]) \neq \emptyset$.

Fifth, the function *diseq* [16] that outputs a set of address accountable for the disequality of two given paths:

$$diseq([i_1, \ldots, i_n], [j_1, \ldots j_m]) = \begin{cases} \emptyset & \text{if } n = m = 0 \\ \{i_1\} & \text{if } n > 0 \text{ and } m = 0 \\ \{j_1\} & \text{if } n = 0 \text{ and } m > 0 \\ \{i_1, j_1\} & \text{if } n, m > 0 \text{ and } i_1 \neq j_1 \\ diseq([i_2, \ldots, i_m], [j_2, \ldots, j_m]) & \text{otherwise} \end{cases}$$

Finally, the function *common* [16] that outputs an element common to two paths (an element that witnesses that $path2set(p) \cap path2set(q) \neq \emptyset$):

$$common([i_1, \ldots, i_n], p) = \begin{cases} \emptyset & \text{if } n = 0 \\ \{i_1\} & \text{if } n > 0 \text{ and } i_1 \in path2set(p) \\ common([i_2, \ldots, i_n], p) & \text{otherwise} \end{cases}$$

Lemma 2 (Finite Model Property). *Let Γ be a conjunction of normalized $\mathsf{TSL_K}$-literals. Let $\overline{e} = |V_{\mathsf{elem}}(\Gamma)|$, $\overline{a} = |V_{\mathsf{addr}}(\Gamma)|$, $\overline{m} = |V_{\mathsf{mem}}(\Gamma)|$, $\overline{p} = |V_{\mathsf{path}}(\Gamma)|$, $\overline{t} = |V_{\mathsf{thid}}(\Gamma)|$ and $\overline{o} = |V_{\mathsf{ord}}(\Gamma)|$. Then the following are equivalent:*
1. *Γ is $\mathsf{TSL_K}$-satisfiable;*
2. *Γ is true in a $\mathsf{TSL_K}$ interpretation $\mathcal{B}$ such that*

$$|\mathcal{B}_{\mathsf{addr}}| \leq \overline{a} + 1 + \overline{m}\,\overline{a}\,\mathsf{K} + \overline{p}^2 + \overline{p}^3 + (\mathsf{K}+2)\overline{m}\,\overline{p}$$
$$|\mathcal{B}_{\mathsf{elem}}| \leq \overline{e} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}|$$
$$|\mathcal{B}_{\mathsf{thid}}| \leq \overline{t} + \mathsf{K}\,\overline{m}\,|\mathcal{B}_{\mathsf{addr}}| + 1$$
$$|\mathcal{B}_{\mathsf{level_K}}| \leq \mathsf{K}$$
$$|\mathcal{B}_{\mathsf{ord}}| \leq \overline{o} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}|$$

Proof. $(2 \to 1)$ is immediate.

$(1 \to 2)$. We prove this implication only for the new $\mathsf{TSL_K}$-literals.

Bearing in mind the auxiliary functions we have defined, let now $\mathcal{A}$ be a $\mathsf{TSL_K}$-interpretation satisfying a set of normalized $\mathsf{TSL_K}$-literals Γ. We use $\mathcal{A}$ to construct a $\mathsf{TSL_K}$-interpretation $\mathcal{B}$ which satisfies Γ.

$$
\begin{aligned}
\mathcal{B}_{\mathsf{level_K}} &= \mathcal{A}_{\mathsf{level_K}} = [0 \dots K-1] \\
\mathcal{B}_{\mathsf{addr}} &= V^{\mathcal{A}}_{\mathsf{addr}} \cup \left\{ null^{\mathcal{A}} \right\} \ \cup \\
&\quad \left\{ m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}(l) \mid m \in V_{\mathsf{mem}}, a \in V_{\mathsf{addr}} \text{ and } l \in \mathcal{B}_{\mathsf{level_K}} \right\} \cup \\
&\quad \left\{ v \in diseq(p^{\mathcal{A}}, q^{\mathcal{A}}) \mid \text{ the literal } p \neq q \text{ is in } \Gamma \right\} \ \cup \\
&\quad \left\{ v \in common(p_1{}^{\mathcal{A}}, p_2{}^{\mathcal{A}}) \mid \text{ the literal } \neg append(p_1, p_2, p_3) \text{ is in } \Gamma \text{ and} \right. \\
&\qquad\qquad\qquad\qquad \left. path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) \neq \emptyset \right\} \ \cup \\
&\quad \left\{ v \in common(p_1{}^{\mathcal{A}} \circ p_2{}^{\mathcal{A}}, p_3{}^{\mathcal{A}}) \mid \text{ the literal } \neg append(p_1, p_2, p_3) \text{ is in } \Gamma \text{ and} \right. \\
&\qquad\qquad\qquad\qquad\qquad \left. path2set^{\mathcal{A}}(p_1{}^{\mathcal{A}}) \cap path2set^{\mathcal{A}}(p_2{}^{\mathcal{A}}) = \emptyset \right\} \ \cup \\
&\quad \left\{ v \in fstL(m^{\mathcal{A}}, p^{\mathcal{A}}, l) \mid fstlock(m, p, l) \text{ is in } \Gamma \right\} \\
&\quad \left\{ v \in unordered(m^{\mathcal{A}}, p^{\mathcal{A}}) \mid \neg ordList(m, p) \text{ is in } \Gamma \right\} \\
\mathcal{B}_{\mathsf{thid}} &= V^{\mathcal{A}}_{\mathsf{thid}} \cup \left\{ \oslash \right\} \cup \left\{ m^{\mathcal{A}}(v^{\mathcal{A}}).lockid^{\mathcal{A}}(l) \mid m \in V_{\mathsf{mem}}, v \in \mathcal{B}_{\mathsf{addr}} \text{ and } l \in \mathcal{B}_{\mathsf{level_K}} \right\} \\
\mathcal{B}_{\mathsf{elem}} &= V^{\mathcal{A}}_{\mathsf{elem}} \cup \left\{ m^{\mathcal{A}}(v).data^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } v \in \mathcal{B}_{\mathsf{addr}} \right\} \\
\mathcal{B}_{\mathsf{ord}} &= V^{\mathcal{A}}_{\mathsf{ord}} \cup \left\{ m^{\mathcal{A}}(v).key^{\mathcal{A}} \mid m \in V_{\mathsf{mem}} \text{ and } v \in \mathcal{B}_{\mathsf{addr}} \right\}
\end{aligned}
$$

These domains satisfy the cardinality constrains expressed in the statement of the theorem. The interpretations of the symbols are:

$$
\begin{aligned}
error^{\mathcal{B}} &= error^{\mathcal{A}} && \\
null^{\mathcal{B}} &= null^{\mathcal{A}} && \\
e^{\mathcal{B}} &= e^{\mathcal{A}} && \text{for each } e \in V_{\mathsf{elem}} \\
a^{\mathcal{B}} &= a^{\mathcal{A}} && \text{for each } a \in V_{\mathsf{addr}} \\
c^{\mathcal{B}} &= c^{\mathcal{A}} && \text{for each } c \in V_{\mathsf{cell}} \\
t^{\mathcal{B}} &= t^{\mathcal{A}} && \text{for each } t \in V_{\mathsf{thid}} \\
k^{\mathcal{B}} &= k^{\mathcal{A}} && \text{for each } k \in V_{\mathsf{ord}} \\
l^{\mathcal{B}} &= l^{\mathcal{A}} && \text{for each } l \in V_{\mathsf{level_K}} \\
m^{\mathcal{B}}(v) &= \big(m^{\mathcal{A}}(v).data^{\mathcal{A}}, m^{\mathcal{A}}(v).key^{\mathcal{A}}, && \text{for each } m \in V_{\mathsf{mem}} \\
&\qquad first_{\mathsf{K}}(m^{\mathcal{A}}, v, 0 \qquad , \mathcal{B}_{\mathsf{addr}}), && \quad \text{and } v \in \mathcal{B}_{\mathsf{addr}} \\
&\qquad \dots && \\
&\qquad first_{\mathsf{K}}(m^{\mathcal{A}}, v, \mathsf{K}-1, \mathcal{B}_{\mathsf{addr}}), && \\
&\qquad m^{\mathcal{A}}(v).lockid^{\mathcal{A}}[0], \dots, m^{\mathcal{A}}(v).lockid^{\mathcal{A}}[\mathsf{K}-1] \big) && \\
s^{\mathcal{B}} &= s^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{addr}} && \text{for each } s \in V_{\mathsf{set}} \\
g^{\mathcal{B}} &= g^{\mathcal{A}} \cap \mathcal{B}_{\mathsf{thid}} && \text{for each } g \in V_{\mathsf{setth}} \\
r^{\mathcal{B}} &= r^{\mathcal{A}} \cap (\mathcal{B}_{\mathsf{addr}} \times \mathcal{B}_{\mathsf{level_K}}) && \text{for each } r \in V_{\mathsf{mrgn}} \\
p^{\mathcal{B}} &= compress(p^{\mathcal{A}}, \mathcal{B}_{\mathsf{addr}}) && \text{for each } p \in V_{\mathsf{path}}
\end{aligned}
$$

Essentially, all variables and constants in $\mathcal{B}$ are interpreted as in $\mathcal{A}$ except that *next* pointers use $first_{\mathsf{K}}$ to point to the next reachable element that has been preserved in $\mathcal{B}_{\mathsf{addr}}$, and paths filter out all elements except those in $\mathcal{B}_{\mathsf{addr}}$. It can be routinely checked that $\mathcal{B}$ is an interpretation of Γ. So it remains to be seen that $\mathcal{B}$ satisfies all literals in Γ assuming that $\mathcal{A}$ does, concluding that $\mathcal{B}$ is indeed a model of Γ. This check is performed by cases. The proof that $\mathcal{B}$ satisfies all TSL_K-literals in Γ is not shown here. We just focus on the new functions and predicates that are not part of TLL. The proof for the missing literals can be found in [16]. For TSL_K-literals we must consider the following cases:

Literals of the form $l_1 \neq l_2$, $k_1 \neq k_2$ and $k_1 \preceq k_2$. Immediate
Literals of the form $c = mkcell(e, k, a_0, \ldots, a_{\mathsf{K}-1}, t_0, \ldots, t_{\mathsf{K}-1})$.

$$\begin{aligned} c^{\mathcal{B}} = c^{\mathcal{A}} &= \left(e^{\mathcal{A}}, k^{\mathcal{A}}, {a_0}^{\mathcal{A}}, \ldots, {a_{\mathsf{K}-1}}^{\mathcal{A}}, {t_0}^{\mathcal{A}}, \ldots, {t_{\mathsf{K}-1}}^{\mathcal{A}}\right) \\ &= \left(e^{\mathcal{B}}, k^{\mathcal{B}}, {a_0}^{\mathcal{B}}, \ldots, {a_{\mathsf{K}-1}}^{\mathcal{B}}, {t_0}^{\mathcal{B}}, \ldots, {t_{\mathsf{K}-1}}^{\mathcal{B}}\right) \end{aligned}$$

Literals of the form $c = rd(m, a)$. In this case we have that

$$\begin{aligned} \left[rd(m,a)\right]^{\mathcal{B}} &= m^{\mathcal{B}}(a^{\mathcal{B}}) \\ &= m^{\mathcal{B}}(a^{\mathcal{A}}) \\ &= \Big(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, m^{\mathcal{A}}(a^{\mathcal{A}}).key^{\mathcal{A}}, \\ &\qquad first_{\mathsf{K}}(m^{\mathcal{A}}, a^{\mathcal{A}}, 0, \mathcal{B}_{sAddr}), \ldots, first_{\mathsf{K}}(m^{\mathcal{A}}, a^{\mathcal{A}}, \mathsf{K}-1, \mathcal{B}_{sAddr}), \\ &\qquad m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}[0], \ldots, m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}[\mathsf{K}-1]\Big) \\ &= \Big(m^{\mathcal{A}}(a^{\mathcal{A}}).data^{\mathcal{A}}, m^{\mathcal{A}}(a^{\mathcal{A}}).key^{\mathcal{A}}, \\ &\qquad m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}(0), \ldots, m^{\mathcal{A}}(a^{\mathcal{A}}).next^{\mathcal{A}}(\mathsf{K}-1), \\ &\qquad m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}[0], \ldots, m^{\mathcal{A}}(a^{\mathcal{A}}).lockid^{\mathcal{A}}[\mathsf{K}-1]\Big) && \text{(Lemma 3)} \\ &= m^{\mathcal{A}}(a^{\mathcal{A}}) \\ &= c^{\mathcal{A}} \\ &= c^{\mathcal{B}} \end{aligned}$$

Literals of the form $g = \{t\}_T$. We have that

$$g^{\mathcal{B}} = g^{\mathcal{A}} \cap \mathcal{B}_{sThId} = \{t^{\mathcal{A}}\}_T \cap \mathcal{B}_{sThId} = \{t^{\mathcal{B}}\}_T \cap \mathcal{B}_{sThId} = \{t^{\mathcal{B}}\}_T$$

Literals of the form $g_1 = g_2 \cup_T g_3$. In this case we have that

$$\begin{aligned} g_1^{\mathcal{B}} = g_1^{\mathcal{A}} \cap \mathcal{B}_{sThId} &= \left(g_2^{\mathcal{A}} \cup_T g_3^{\mathcal{A}}\right) \cap \mathcal{B}_{sThId} \\ &= \left(g_2^{\mathcal{A}} \cap \mathcal{B}_{sThId}\right) \cup_T \left(g_3^{\mathcal{A}} \cap \mathcal{B}_{sThId}\right) \\ &= g_2^{\mathcal{B}} \cup_T g_3^{\mathcal{B}} \end{aligned}$$

Literals of the form $g_1 = g_2 \setminus_T g_3$. We have that

$$\begin{aligned} g_1^{\mathcal{B}} = g_1^{\mathcal{A}} \cap \mathcal{B}_{sThId} &= \left(g_2^{\mathcal{A}} \setminus_T g_3^{\mathcal{A}}\right) \cap \mathcal{B}_{sThId} \\ &= \left(g_2^{\mathcal{A}} \cap \mathcal{B}_{sThId}\right) \setminus_T \left(g_3^{\mathcal{A}} \cap \mathcal{B}_{sThId}\right) \\ &= g_2^{\mathcal{B}} \setminus_T g_3^{\mathcal{B}} \end{aligned}$$

Literals of the form $r = \langle a, l \rangle_{\mathsf{mr}}$. We have that

$$
\begin{aligned}
r^{\mathcal{B}} &= r^{\mathcal{A}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{\mathsf{level_K}}) \\
&= \langle a^{\mathcal{A}}, l^{\mathcal{A}} \rangle_{\mathsf{mr}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{\mathsf{level_K}}) \\
&= \langle a^{\mathcal{B}}, l^{\mathcal{B}} \rangle_{\mathsf{mr}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{\mathsf{level_K}}) \\
&= \langle a^{\mathcal{B}}, l^{\mathcal{B}} \rangle_{\mathsf{mr}}
\end{aligned}
$$

Literals of the form $r_1 = r_2 \cup_{\mathsf{mr}} r_3$. In this case we have that

$$
\begin{aligned}
r_1^{\mathcal{B}} &= r_1^{\mathcal{A}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{\mathsf{level_K}}) \\
&= \left(r_2^{\mathcal{A}} \cup_{\mathsf{mr}} r_3^{\mathcal{A}}\right) \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{\mathsf{level_K}}) \\
&= \left(r_2^{\mathcal{A}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{\mathsf{level_K}})\right) \cup_{\mathsf{mr}} \left(r_3^{\mathcal{A}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{\mathsf{level_K}})\right) \\
&= r_2^{\mathcal{B}} \cup_{\mathsf{mr}} r_3^{\mathcal{B}}
\end{aligned}
$$

Literals of the form $r_1 = r_2 -_{\mathsf{mr}} r_3$. We have that

$$
\begin{aligned}
r_1^{\mathcal{B}} &= r_1^{\mathcal{A}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{sLevelK}) \\
&= \left(r_2^{\mathcal{A}} -_{\mathsf{mr}} r_3^{\mathcal{A}}\right) \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{sLevelK}) \\
&= \left(r_2^{\mathcal{A}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{sLevelK})\right) -_{\mathsf{mr}} \left(r_3^{\mathcal{A}} \cap (\mathcal{B}_{sAddr} \times \mathcal{B}_{sLevelK})\right) \\
&= r_2^{\mathcal{B}} -_{\mathsf{mr}} r_3^{\mathcal{B}}
\end{aligned}
$$

Literals of the form $s = \mathit{addr2set}_{\mathsf{K}}(m, a, l)$. Let $x = a^{\mathcal{B}} = a^{\mathcal{A}}$. Then, we have that

$$
\begin{aligned}
s^{\mathcal{B}} &= s^{\mathcal{A}} \cap \mathcal{B}_{sAddr} \\
&= \left\{ y \in \mathcal{A}_{\mathsf{addr}} \mid \exists\, p \in \mathcal{A}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{A}}, x, y, l, p) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}} \right\} \cap \mathcal{B}_{sAddr} \\
&= \left\{ y \in \mathcal{B}_{sAddr} \mid \exists\, p \in \mathcal{A}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{A}}, x, y, l, p) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}} \right\} \\
&= \left\{ y \in \mathcal{B}_{sAddr} \mid \exists\, p \in \mathcal{B}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{B}}, x, y, l, p) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{B}} \right\}
\end{aligned}
$$

It just remains to see that the last equality holds. Let

- $S_{\mathcal{B}} = \left\{ y \in \mathcal{B}_{sAddr} \mid \exists\, p \in \mathcal{B}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{B}}, x, y, l, p) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{B}} \right\}$, and
- $S_{\mathcal{A}} = \left\{ y \in \mathcal{B}_{sAddr} \mid \exists\, p \in \mathcal{A}_{\mathsf{path}}\ s.t.,\ (m^{\mathcal{A}}, x, y, l, p) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}} \right\}$

We first show that $S_{\mathcal{A}} \subseteq S_{\mathcal{B}}$. Let $y \in S_{\mathcal{A}}$. Then exists $p \in \mathcal{A}_{\mathsf{path}}$ such that $(m^{\mathcal{A}}, x, y, l, p) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}}$. Then, by definition of $\mathit{reach}_{\mathsf{K}}$ there are two possible cases.

- If $p = \epsilon$ and $x = y$, then $(m^{\mathcal{B}}, x, y, l, \epsilon^{\mathcal{B}}) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{B}}$ and therefore $y \in S_{\mathcal{B}}$.
- Otherwise, there exists $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ s.t.,

$$\begin{array}{ll} i)\ p = [a_1, \ldots, a_n] & iii)\ m^{\mathcal{A}}(a_r).next^{\mathcal{A}}(l) = a_{r+1}, \text{for } 1 \leq r < n \\ ii)\ x = a_1 & iv)\ m^{\mathcal{A}}(a_n).next^{\mathcal{A}}(l) = y \end{array}$$

Then, we only need to find $\tilde{a}_1, \ldots, \tilde{a}_m \in \mathcal{B}_{\mathsf{addr}}$ s.t.,

$$\begin{array}{ll} i)\ q = [\tilde{a}_1, \ldots, \tilde{a}_m] & iii)\ m^{\mathcal{B}}(\tilde{a}_r).next^{\mathcal{B}}(l) = \tilde{a}_{r+1}, \text{for } 1 \leq r < m \\ ii)\ x = \tilde{a}_1 & iv)\ m^{\mathcal{B}}(\tilde{a}_m).next^{\mathcal{B}}(l) = y \end{array}$$

We define $\tilde{a}_1 = a_1 = x$ and $\tilde{a}_2 = \mathit{first}_{\mathsf{K}}(m^{\mathcal{A}}, \tilde{a}_1, l, \mathcal{B}_{sAddr})$. Then we know that $\tilde{a}_2 = m^{\mathcal{B}}(\tilde{a}_1).next^{\mathcal{B}}(l)$ and that $\tilde{a}_2 \in \mathcal{B}_{sAddr}$ and thus $\tilde{a}_2 \in \mathcal{B}_{\mathsf{addr}}$. Then, if $\tilde{a}_2 = y$ there is nothing else to prove. On the other hand, if $\tilde{a}_2 \neq y$ then we proceed in the same way to define $\tilde{a}_3$ and so on until $\tilde{a}_{m+1} = y$. Notice that this way, y is guaranteed to be found in at most n steps.

To show that $S_{\mathcal{B}} \subseteq S_{\mathcal{A}}$ we proceed in a similar way. Let $y \in S_{\mathcal{B}}$. Then $x = y$ and $p = \epsilon$ and thus $(m^{\mathcal{A}}, x, y, l, \epsilon^{\mathcal{A}}) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}}$, or exists $a_1, \ldots, a_n \in \mathcal{B}_{sAddr}$ such that

$$\begin{array}{ll} i)\ p = [a_1, \ldots, a_n] & iii)\ m^{\mathcal{B}}(a_r).next^{\mathcal{B}}(l) = a_{r+1}, \text{for } 1 \leq r < n \\ ii)\ x = a_1 & iv)\ m^{\mathcal{B}}(a_n).next^{\mathcal{B}}(l) = y \end{array}$$

As we know that $a_1, \ldots, a_n, y \in \mathcal{B}_{sAddr}$, by definition of $\mathit{first}_{\mathsf{K}}$ we know that exists $s \geq 1$ s.t.,

$$m^{\mathcal{A}}\Big(\cdots\big(m^{\mathcal{A}}\,(a_1)\,.\underbrace{next^{\mathcal{A}}\,(l)\,\big)\cdots\Big).next^{\mathcal{A}}\,(l)}_{s} = a_2$$

Let then $a_1^1, \ldots, a_1^{s-1} \in \mathcal{A}_{\mathsf{addr}}$ such that

$$\begin{aligned} m^{\mathcal{A}}(a_1).next^{\mathcal{A}}(l) &= a_1^1 \\ m^{\mathcal{A}}(a_1^1).next^{\mathcal{A}}(l) &= a_1^2 \\ &\vdots \\ m^{\mathcal{A}}(a_1^{s-1}).next^{\mathcal{A}}(l) &= a_2 \end{aligned}$$

We then use $a_1^1, \ldots, a_1^{s-1}$ to construct the section of a path q that goes from a_1 up to a_2. Finally we use the same approach to finish with the construction of such path in $\mathcal{A}$. Then we have that $(m^{\mathcal{A}}, x, y, l, q^{\mathcal{A}}) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}}$. Then, $y \in S_{\mathcal{A}}$.

Literals of the form $p = \mathit{getp}_{\mathsf{K}}(m, a, b, l)$. We consider two possible cases.

– Case $b^{\mathcal{A}} \in \mathit{addr2set}_{\mathsf{K}}(m^{\mathcal{A}}, a^{\mathcal{A}}, l)$.
Since $(m^{\mathcal{A}}, a^{\mathcal{A}}, b^{\mathcal{A}}, l, p^{\mathcal{A}}) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}}$, it is enough to prove:

$$(m^{\mathcal{A}}, x, y, l, q) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}} \quad \rightarrow \quad (m^{\mathcal{B}}, x, y, l, \mathit{compress}(q, \mathcal{B}_{sAddr})) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{B}}$$

for each $x, y \in \mathcal{B}_{sAddr}$ and $q \in \mathcal{A}_{\mathsf{path}}$. Assume that $(m^{\mathcal{A}}, x, y, l, q) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}}$. If $x = y$ and $q = \epsilon$, then $(m^{\mathcal{B}}, x, y, l, \mathit{compress}(q, \mathcal{B}_{sAddr})) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{B}}$. Otherwise, there exists $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ such that:

$$\begin{array}{ll} i)\ q = [a_1, \ldots, a_n] & iii)\ m^{\mathcal{A}}(a_r).\mathit{next}^{\mathcal{A}}(l) = a_{r+1}, \text{for } 1 \leq r < n \\ ii)\ x = a_1 & iv)\ m^{\mathcal{A}}(a_n).\mathit{next}^{\mathcal{A}}(l) = y \end{array}$$

Then, we proceed by induction on n.

- If $n = 1$, then $q = [a_1]$ and therefore $\mathit{compress}(q, \mathcal{B}_{sAddr}) = [a_1]$, since $x = a_1 \in \mathcal{B}_{sAddr}$. Besides $m^{\mathcal{A}}(a_1).\mathit{next}^{\mathcal{A}}(l) = y$ which implies that $m^{\mathcal{B}}(a_1).\mathit{next}^{\mathcal{B}}(l) = y$. Then $(m^{\mathcal{B}}, x, y, l, \mathit{compress}(q, \mathcal{B}_{sAddr})) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{B}}$.
- If $n > 1$, then let $a_i = \mathit{first}_{\mathsf{K}}(m^{\mathcal{A}}, x, l, \mathcal{B}_{sAddr})$. As

$$q = [x = a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n]$$

we have that

$$\mathit{compress}(q, \mathcal{B}_{sAddr}) = [x = a_1] \circ \mathit{compress}([a_i, a_{i+1}, \ldots, a_n], \mathcal{B}_{sAddr})$$

Besides, as $(m^{\mathcal{A}}, a_i, y, l, [a_i, a_{i+1}, \ldots, a_n]) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{A}}$, by induction we have that

$$(m^{\mathcal{B}}, a_i, y, l, \mathit{compress}([a_i, a_{i+1}, \ldots, a_n], \mathcal{B}_{sAddr})) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{B}}$$

Moreover $m^{\mathcal{B}}(x).\mathit{next}^{\mathcal{B}}(l) = a_i$ and therefore

$$(m^{\mathcal{B}}, x, y, l, \mathit{compress}(q, \mathcal{B}_{sAddr})) \in \mathit{reach}_{\mathsf{K}}{}^{\mathcal{B}}$$

– Case $b^{\mathcal{A}} \notin \mathit{addr2set}_{\mathsf{K}}(m^{\mathcal{A}}, a^{\mathcal{A}}, l)$.
In such case we have that $p^{\mathcal{A}} = \epsilon$, which implies that $p^{\mathcal{B}} = \epsilon$. Then using a reasoning similar to the previous case we can deduce that $b^{\mathcal{B}} \notin \mathit{addr2set}_{\mathsf{K}}(m^{\mathcal{B}}, a^{\mathcal{B}}, l)$.

Literals of the form $a = \mathit{fstlock}_{\mathsf{K}}(m, p, l)$. If we consider the case $p = \epsilon$, then we know that $\mathit{fstlock}_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, \epsilon^{\mathcal{A}}, l^{\mathcal{A}}) = \mathit{null}^{\mathcal{A}}$. At the same time, we know that $\epsilon^{\mathcal{B}} = \mathit{compress}(\epsilon^{\mathcal{A}}, \mathcal{B}_{sAddr})$ and so $\mathit{fstlock}_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, \epsilon^{\mathcal{B}}, l^{\mathcal{B}}) = \mathit{null}^{\mathcal{B}}$. Let's now consider the case at which $p = [a_1, \ldots, a_n]$. There are two scenarios to consider.

– If for all $1 \leq k \leq n$, $m^{\mathcal{A}}(a_k^{\mathcal{A}}).lockid(l) = \oslash$, then we have that

$$fstlock_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, p^{\mathcal{A}}, l^{\mathcal{A}}) = null^{\mathcal{A}}$$

Notice that function *compress* returns a subset of the path it receives with the property that all addresses in the returned path belong to the received set. Then, if $[\tilde{a}_1, \ldots, \tilde{a}_m] = p^{\mathcal{B}} = compress(p^{\mathcal{A}}, \mathcal{B}_{sAddr})$, we know that $\{\tilde{a}_1, \ldots, \tilde{a}_m\} \subseteq \mathcal{B}_{sAddr}$ and therefore for all $1 \leq j \leq m$, $m^{\mathcal{B}}(\tilde{a}_j).lockid^{\mathcal{B}}(l^{\mathcal{B}}) = \oslash$. Then, we can finally conclude that in fact $fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, p^{\mathcal{B}}, l^{\mathcal{B}}) = null^{\mathcal{B}}$.

– If exists a $1 \leq k \leq n$ such that for all $1 \leq j < k$, $m^{\mathcal{A}}(a_j^{\mathcal{A}}).lockid(l) = \oslash$ and $m^{\mathcal{A}}(a_k^{\mathcal{A}}).lockid(l) \neq \oslash$ then since $a^{\mathcal{B}} = a^{\mathcal{A}}$, we can say that $a^{\mathcal{B}} = a^{\mathcal{A}} = x \in \mathcal{B}_{sAddr}$. It then remains to verify whether

$$x = fstlock_{\mathsf{K}}{}^{\mathcal{A}}(m^{\mathcal{A}}, p^{\mathcal{A}}, l^{\mathcal{A}}) \;\rightarrow\; x = fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, compress(p^{\mathcal{A}}, \mathcal{B}_{sAddr}), l^{\mathcal{B}})$$

By definition of $fstlock_{\mathsf{K}}$ we have that $x = a_k^{\mathcal{A}}$ and by κ we know that $a_k^{\mathcal{A}} \in \mathcal{B}_{sAddr}$. Let $[\tilde{a}_1, \ldots, \tilde{a}_i, \ldots, \tilde{a}_m] = compress(p^{\mathcal{A}}, \mathcal{B}_{sAddr})$ such that $\tilde{a}_i = a_k^{\mathcal{A}}$. We also know that $\tilde{a}_j \in \mathcal{B}_{sAddr}$ for all $1 \leq j \leq m$. Then, as *compress* preserves the order and for all $1 \leq j < k$, $m^{\mathcal{A}}(a_j^{\mathcal{A}}).lockid^{\mathcal{A}}(l^{\mathcal{A}}) = \oslash$, we have that for all $1 \leq j < i$, $m^{\mathcal{B}}(\tilde{a}_j).lockid^{\mathcal{B}}(l^{\mathcal{B}}) = \oslash$. Besides $m^{\mathcal{B}}(\tilde{a}_i).lockid^{\mathcal{B}}(l^{\mathcal{B}}) \neq \oslash$. Then:

$$\begin{aligned} fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, compress(p^{\mathcal{A}}), l^{\mathcal{B}}) &= fstlock_{\mathsf{K}}{}^{\mathcal{B}}(m^{\mathcal{B}}, [\tilde{a}_1, \ldots, \tilde{a}_m], l^{\mathcal{B}}) \\ &= \tilde{a}_i \\ &= a^{\mathcal{A}} \\ &= x \end{aligned}$$

Literals of the form $ordList(m, p)$**.** Assume that $(m^{\mathcal{A}}, p^{\mathcal{A}}) \in ordList^{\mathcal{A}}$. We want to see that $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordList^{\mathcal{B}}$ i.e., $(m^{\mathcal{B}}, compress(p^{\mathcal{A}}, \mathcal{B}_{sAddr})) \in ordList^{\mathcal{B}}$. We proceed by induction on p.

– If $p = \epsilon$, by definition of *compress* and *ordList*, we have that $(m^{\mathcal{B}}, \epsilon^{\mathcal{B}}) \in ordList^{\mathcal{B}}$.

– If $p = [a_1]$, we know that $(m^{\mathcal{A}}, [a_1]^{\mathcal{A}}) \in ordList^{\mathcal{A}}$ and that $p^{\mathcal{B}} = compress(p^{\mathcal{A}}, \mathcal{B}_{sAddr})$. Then, if $a_1^{\mathcal{A}} \in \mathcal{B}_{sAddr}$, we have that $p^{\mathcal{B}} = [a_1]^{\mathcal{B}}$ and then clearly $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordList^{\mathcal{B}}$ holds. On the other hand, if $a_1^{\mathcal{A}} \notin \mathcal{B}_{sAddr}$, then $p^{\mathcal{B}} = \epsilon^{\mathcal{B}}$ and once more $(m^{\mathcal{B}}, p^{\mathcal{B}}) \in ordList^{\mathcal{B}}$ holds.

– If $p = [a_1, \ldots, a_{n+1}]$ with $n \geq 1$, then we have two possible cases to bear in mind. If we consider the case at which $a_1^{\mathcal{A}} \notin \mathcal{B}_{sAddr}$ then $compress(p^{\mathcal{A}}, \mathcal{B}_{sAddr}) = compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{sAddr})$ and as by induction we have that $(m^{\mathcal{B}}, compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{sAddr})) \in ordList^{\mathcal{B}}$ we conclude that $(m^{\mathcal{B}}, compress([a_1, a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{sAddr})) \in ordList^{\mathcal{B}}$. On the other hand, if $a_1^{\mathcal{A}} \in \mathcal{B}_{sAddr}$ then once more, by induction, $(m^{\mathcal{B}}, compress([a_2, \ldots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{sAddr})) \in ordList^{\mathcal{B}}$. Besides, as we have that $m^{\mathcal{A}}(a_1^{\mathcal{A}}).key^{\mathcal{A}} \preceq m^{\mathcal{A}}(a_2^{\mathcal{A}}).key^{\mathcal{A}}$ we can deduce that $m^{\mathcal{B}}(a_1^{\mathcal{A}}).key^{\mathcal{B}} \preceq m^{\mathcal{B}}(a_2^{\mathcal{A}}).key^{\mathcal{B}}$. And so, $(m^{\mathcal{B}},\ \mathit{compress}([a_1, a_2, \dots, a_{n+1}]^{\mathcal{A}}, \mathcal{B}_{sAddr})) \in \mathit{ordList}^{\mathcal{B}}$.

Literals of the form $\neg \mathit{ordList}(m, p)$. Let's assume that $(m^{\mathcal{A}}, p^{\mathcal{A}}) \notin \mathit{ordList}^{\mathcal{A}}$. We want to see that $(m^{\mathcal{B}}, p^{\mathcal{B}}) \notin \mathit{ordList}^{\mathcal{B}}$. If $(m^{\mathcal{A}}, p^{\mathcal{A}}) \notin \mathit{ordList}^{\mathcal{A}}$, then it means that $p = [a_1, \dots, a_n]$ with $n \geq 2$ and $m^{\mathcal{A}}(a_{i+1}).key^{\mathcal{A}} \preceq m^{\mathcal{A}}(a_i).key^{\mathcal{A}}$ and $m^{\mathcal{A}}(a_{i+1}).key^{\mathcal{A}} \neq m^{\mathcal{A}}(a_i).key^{\mathcal{A}}$ for some $i \in 1, \dots, n-1$. Let that i be the one such that for all $j < i$, $m^{\mathcal{A}}(a_j).key^{\mathcal{A}} \preceq m^{\mathcal{A}}(a_{j+1}).key^{\mathcal{A}}$. Then, by Lemma 4 we know that $\mathit{unordered}(m^{\mathcal{A}}, [a_1, \dots, a_n]^{\mathcal{A}}) \neq \emptyset$ and besides $\{a_i^{\mathcal{A}}, a_{i+1}^{\mathcal{A}}\} \subseteq \mathit{unordered}(m^{\mathcal{A}}, [a_1, \dots, a_n]^{\mathcal{A}}) \subseteq \mathcal{B}_{sAddr}$. This means that $\mathit{compress}([a_1, \dots, a_n]^{\mathcal{A}}, \mathcal{B}_{sAddr}) = [\tilde{a}_1, \dots, a_i, a_{i+1}, \dots, \tilde{a}_m]^{\mathcal{B}}$. Therefore, since $m^{\mathcal{B}}(a_{i+1}).key^{\mathcal{B}} \preceq m^{\mathcal{B}}(a_i).key^{\mathcal{B}}$ and $m^{\mathcal{B}}(a_{i+1}).key^{\mathcal{B}} \neq m^{\mathcal{B}}(a_i).key^{\mathcal{B}}$, we have that $(m^{\mathcal{B}}, \mathit{compress}([a_1, \dots, a_n]^{\mathcal{A}}, \mathcal{B}_{sAddr})) \notin \mathit{ordList}^{\mathcal{B}}$.

B Missing Implementations

```
procedure Search(SkipList sl, Value v)
    int i := K − 1                          //@ mrgn m_r := ∅
    Node* pred := sl.head
    pred.locks[i].lock()                    //@ m_r := m_r ∪ {(pred, i)}
    Node* curr := pred.next[i]
    curr.locks[i].lock()                    //@ m_r := m_r ∪ {(curr, i)}
    while 0 ≤ i ∧ curr.val ≠ v do
        if i < K − 1 then
            pred.locks[i].lock()            //@ m_r := m_r ∪ {(pred, i)}
            curr := pred.next[i]
            curr.locks[i].lock()            //@ m_r := m_r ∪ {(curr, i)}
            pred.next[i + 1].locks[i + 1].unlock()   //@ m_r := m_r − {(pred.next[i + 1], i + 1)}
            pred.locks[i + 1].unlock()      //@ m_r := m_r − {(pred, i + 1)}
        end if
        while curr.val < v do
            pred.locks[i].unlock()          //@ m_r := m_r − {(pred, i)}
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()            //@ m_r := m_r ∪ {(curr, i)}
        end while
        i := i − 1
    end while
    Bool valueIsIn := (curr.val = v)
    if i = K − 1 then
        curr.locks[i].unlock()              //@ m_r := m_r − {(curr, i)}
        pred.locks[i].unlock()              //@ m_r := m_r − {(pred, i)}
    else
        curr.locks[i + 1].unlock()          //@ m_r := m_r − {(curr, i + 1)}
        pred.locks[i + 1].unlock()          //@ m_r := m_r − {(pred, i + 1)}
    end if
    return valueIsIn
end procedure
```

Fig. 7. Algorithm for searching on a concurrent lock-coupling skiplist

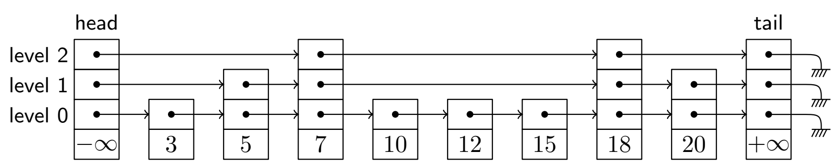
```
procedure Remove(SkipList sl, Value v)
    Vector < Node* > upd[0..K − 1]              //@ mrgn m_r := ∅
    Node* pred := sl.head
    pred.locks[K − 1].lock()                     //@ m_r := m_r ∪ {(pred, K)}
    Node* curr := pred.next[K − 1]
    curr.locks[K − 1].lock()                     //@ m_r := m_r ∪ {(curr, K)}
    for i := K − 1 downto 0 do
        if i < K − 1 then
            pred.locks[i].lock()                 //@ m_r := m_r ∪ {(pred, i)}
            curr := pred.next[i]
            curr.locks[i].lock()                 //@ m_r := m_r ∪ {(curr, i)}
        end if
        while curr.val < v do
            pred.locks[i].unlock()               //@ m_r := m_r − {(pred, i)}
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()                 //@ m_r := m_r ∪ {(curr, i)}
        end while
        upd[i] := pred
    end for
    for i := K − 1 downto 0 do
        if upd[i].next[i] = curr ∧ curr.val = v then
            upd[i].next[i] := curr.next[i]       //@ sl.r := sl.r − {(curr, i)}
            curr.locks[i].unlock()               //@ m_r := m_r − {(curr, i)}
        else
            upd[i].next[i].locks[i].unlock()     //@ m_r := m_r − {upd[i].next[i], i)}
        end if
        upd[i].locks[i].unlock()                 //@ m_r := m_r − {(upd[i], i)}
    end for
    Bool valueWasIn := (curr.val = v)
    if valueWasIn then
        free (curr)
    end if
    return valueWasIn
end procedure
```

Fig. 8. Algorithm for deletion on a concurrent lock-coupling skiplist

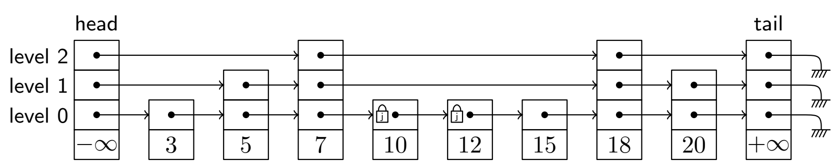
C No Thread Overtakes

In this section we proof that, in fact, no thread can overtake another thread, considering the region of the skiplist that can potentially be modified by the latter one. This does not mean that no thread can overtake another one using a higher level pointer. Instead, we want to verify that no thread can go through the region to be modified by another thread.

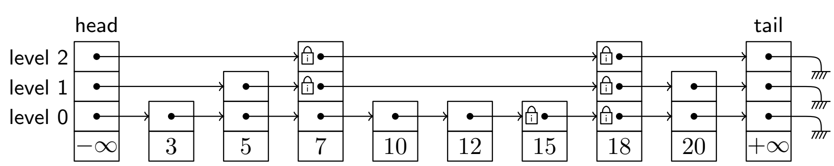
Let's consider the skiplist shown in Fig. 9(a). Imagine a thread j trying to insert a one level node with value 11. Then, after reaching the position where the node must be inserted the skiplist may look as the one depicted in Fig. 9(b). If then another thread, lets say i, wants to insert a node with value 19, it will undoubtedly jump over the nodes locked by thread j. This is not the situation we are trying to prevent, since this is a correct behavior for a concurrent skiplist. Moreover, the modifications introduced by thread i will not interfere with thread i at all.



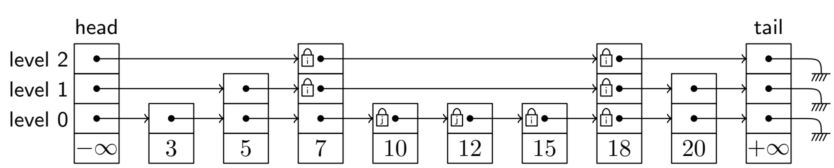
(a) initial skiplist



(b) skiplist before thread j inserts the new node



(c) skiplist after thread i is ready to insert node 16



(d) skiplist with thread j about to insert node 11

Fig. 9. Example of skiplist

However, let's now consider the same skiplist depicted in Fig. 9(a). Imagine now that thread i wants to insert a node of three levels with value 16. In such case, before the insertion is accomplished, the skiplist will have the aspect depicted in Fig. 9(c). Is in this case we want to show that thread j will not be able to progress up to the position where node 11 must be inserted, ending up in a scenario as the one shown in Fig. 9(d). An informal reasoning let us deduce that thread j cannot reach such position since node 7 should be reached in a top-down fashion, something that cannot happen since thread i has locked the upper levels of such node. We now proceed to formalize this reasoning.

We begin by extending the actual code with ghost code to aid the verification. We add the ghost variables L, U, H denoting the limits of the minimum region we are sure a thread can potentially modified (PM). Such region is then formally defined as a masked region by:

$$PM \hat{=} \bigcup_{i=0}^{H} \left\{ (n, i) \mid n \in getp_{\mathsf{K}}(h, L, U, i) \right\}$$

assuming that the skiplist resides in the heap h. Considering once more the situation of thread i trying to insert a node with value 16 depicted in Fig. 9(c), we can represent the PM region of such thread as shown with dashed lines in Fig. 10.

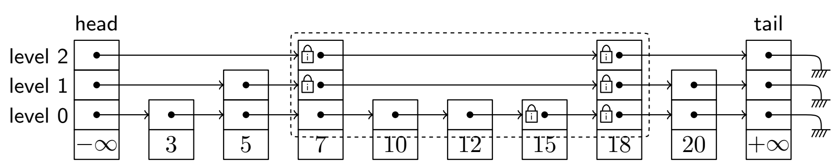


Fig. 10. PM region for thread i when inserting node 16

We first extend the algorithm for *search*, *insert* and *remove* with the new ghost variables. Then, we define a function $skipRgTh : \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \to \mathsf{setth}$ and a predicate $lastTh : \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K}$. Given a lower address L, and upper address U and a level L, the function $skipRgTh(L, U, H)$ returns the set of threads identifiers which has a locked node within the PM region described by L, U and H. Meanwhile, $lastTh(L, U, H)$ holds whenever $skipRgTh(L, U, H) = \emptyset_T$.

The new algorithms extended with ghost variables are depicted in Fig. 11, 12 and 13. Notice that when setting H to -1 we are just saying that PM becomes empty.

The main idea is that ever moment, $PM^{[t]}$ represents the minimum region of the skiplist we are sure thread t can potentially modify. After every transition is taken, we end up with a subregion (possibly the same one) as before. We would like to ensure that every transition, taken by thread$t$ or any other thread of the system, does not increment the number of threads within its PM region.

```
procedure SEARCH(SkipList sl, Value v)
    int i := K − 1                                  //@ mrgn m_r := ∅
                                                    //@ L := sl.head
                                                    //@ U := sl.tail
                                                    //@ H := K − 1
    Node* pred := sl.head
    pred.locks[i].lock()                            //@ m_r := m_r ∪ {(pred, i)}
    Node* curr := pred.next[i]
    curr.locks[i].lock()                            //@ m_r := m_r ∪ {(curr, i)}
    while 0 ≤ i ∧ curr.val ≠ v do
        if i < K − 1 then
            pred.locks[i].lock()                    //@ m_r := m_r ∪ {(pred, i)}
                                                    //@ U := curr
            curr := pred.next[i]
            curr.locks[i].lock()                    //@ m_r := m_r ∪ {(curr, i)}
            pred.next[i + 1].locks[i + 1].unlock()
                                                    //@ m_r := m_r − {(pred.next[i + 1], i + 1)}
            pred.locks[i + 1].unlock()              //@ m_r := m_r − {(pred, i + 1)}
                                                    //@ H := i
        end if
        while curr.val < v do
            pred.locks[i].unlock()                  //@ m_r := m_r − {(pred, i)}
                                                    //@ L := curr
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()                    //@ m_r := m_r ∪ {(curr, i)}
        end while
        i := i − 1
    end while
    Bool valueIsIn := (curr.val = v)
    if i = K − 1 then
        curr.locks[i].unlock()                      //@ m_r := m_r − {(curr, i)}
                                                    //@ U := L
        pred.locks[i].unlock()                      //@ m_r := m_r − {(pred, i)}
                                                    //@ H := -1
    else
        curr.locks[i + 1].unlock()                  //@ m_r := m_r − {(curr, i + 1)}
                                                    //@ U := L
        pred.locks[i + 1].unlock()                  //@ m_r := m_r − {(pred, i + 1)}
                                                    //@ H := -1
    end if
    return valueIsIn
end procedure
```

Fig. 11. Algorithm for searching on a concurrent lock-coupling skiplist

```
procedure Insert(SkipList sl, Value newval)
    Vector⟨Node*⟩ upd[0..K − 1]                  //@ mrgn m_r := ∅
                                                //@ L := sl.head
                                                //@ U := sl.tail
                                                //@ H := K − 1
    lvl := randomLevel(K)
    Node* pred := sl.head
    pred.locks[K − 1].lock()                     //@ m_r := m_r ∪ {(pred, K − 1)}
    Node* curr := pred.next[K − 1]
    curr.locks[K − 1].lock()                     //@ m_r := m_r ∪ {(curr, K − 1)}
    for i := K − 1 downto 0 do
        if i < K − 1 then
            pred.locks[i].lock()                 //@ m_r := m_r ∪ {(pred, i)}
                                                //@ U := curr
            curr := pred.next[i]
            curr.locks[i].lock()                 //@ m_r := m_r ∪ {(curr, i)}
            if i ≥ lvl then
                pred.next[i + 1].locks[i + 1].unlock()
                                    //@ m_r := m_r − {(pred.next[i + 1], i + 1)}
                pred.locks[i + 1].unlock()       //@ m_r := m_r − {(pred, i + 1)}
                                                //@ H := i
            end if
        end if
        while curr.val < newval do
            pred.locks[i].unlock()               //@ m_r := m_r − {(pred, i)}
                                                //@ if (i = lvl){L := curr}
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()                 //@ m_r := m_r ∪ {(curr, i)}
        end while
        upd[i] := pred
    end for
    Bool valueWasIn := (curr.val = newval)
    if valueWasIn then
        for i := 0 to lvl do
            upd[i].next[i].locks[i].unlock()     //@ m_r := m_r − {(upd[i].next[i], i)}
            upd[i].locks[i].unlock()             //@ m_r := m_r − {(upd[i], i)}
        end for
    else
        x := CreateNode(lvl, newval)
        for i := 0 to lvl do
            x.next[i] := upd[i].next[i]
            upd[i].next[i] := x                  //@ sl.r := sl.r ∪ {(x, i)}
            x.next[i].locks[i].unlock()          //@ m_r := m_r − {(x.next[i], i)}
            upd[i].locks[i].unlock()             //@ m_r := m_r − {(upd[i], i)}
        end for
    end if
                                                //@ H = −1
    return ¬valueWasIn
end procedure
```

Fig. 12. Algorithm for insertion on a concurrent lock-coupling skiplist

```
procedure Remove(SkipList sl, Value v)
    Vector < Node* > upd[0..K − 1]              //@ mrgn m_r := ∅
                                                //@ L := sl.head
                                                //@ U := sl.tail
                                                //@ H := K − 1
    Node* pred := sl.head
    pred.locks[K − 1].lock()                    //@ m_r := m_r ∪ {(pred, K)}
    Node* curr := pred.next[K − 1]
    curr.locks[K − 1].lock()                    //@ m_r := m_r ∪ {(curr, K)}
    Node* aux
    for i := K − 1 downto 0 do
        if i < K − 1 then
            pred.locks[i].lock()                //@ m_r := m_r ∪ {(pred, i)}
                                                //@ U := curr
            aux := curr
            curr := pred.next[i]
            curr.locks[i].lock()                //@ m_r := m_r ∪ {(curr, i)}
            if aux.val ≠ e then
                aux.locks[i + 1].unlock()
                pred.locks[i + 1].unlock()      //@ H := i
            end if
        end if
        while curr.val < v do
            pred.locks[i].unlock()              //@ m_r := m_r − {(pred, i)}
                                                //@ if (aux.val ≠ e){L := curr}
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()                //@ m_r := m_r ∪ {(curr, i)}
        end while
        upd[i] := pred
    end for
    for i := K − 1 downto 0 do
        if upd[i].next[i] = curr ∧ curr.val = v then
            upd[i].next[i] := curr.next[i]      //@ sl.r := sl.r − {(curr, i)}
            curr.locks[i].unlock()              //@ m_r := m_r − {(curr, i)}
        else
            upd[i].next[i].locks[i].unlock()    //@ m_r := m_r − {upd[i].next[i], i)}
        end if
        upd[i].locks[i].unlock()                //@ m_r := m_r − {(upd[i], i)}
                                                //@ H := i − 1
    end for
    Bool valueWasIn := (curr.val = v)
    if valueWasIn then
        free (curr)
    end if
    return valueWasIn
end procedure
```

Fig. 13. Algorithm for deletion on a concurrent lock-coupling skiplist

D Non Termination Under Weak Fairness

Here we proof that the implementation given in Fig. 3 does not ensure termination of all threads under the assumption of weak-fairness. For such purpose, consider the skiplist depicted in Fig. 14.

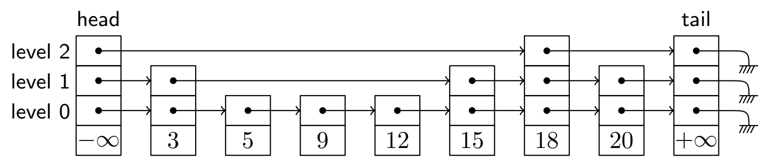
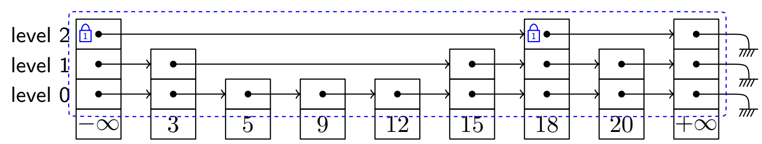
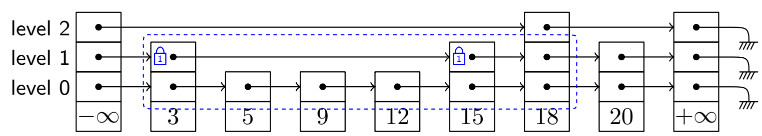


Fig. 14. An example of skiplist

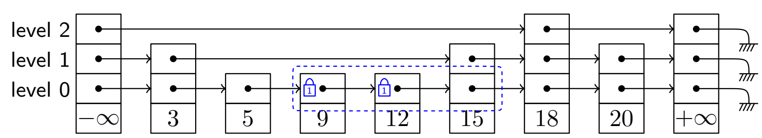
We use values $L^{[t]}$, $U^{[t]}$ and $H^{[t]}$ to denote the section of the skiplist that can be potentially modified by thread t. $L^{[t]}$ describes the lowest address bound while $U^{[t]}$ denotes the upper address bound. Meanwhile, $H^{[t]}$ represents the higher level to be modified. Considering the skiplist at Fig. 14 we consider two thread running concurrently. Thread 1 (called T_1) will insert value 14 with height 1, while thread 2 (denoted T_2) inserts value 16 with height 2.



(a) initial locking by T_1



(b) T_1 on level 1

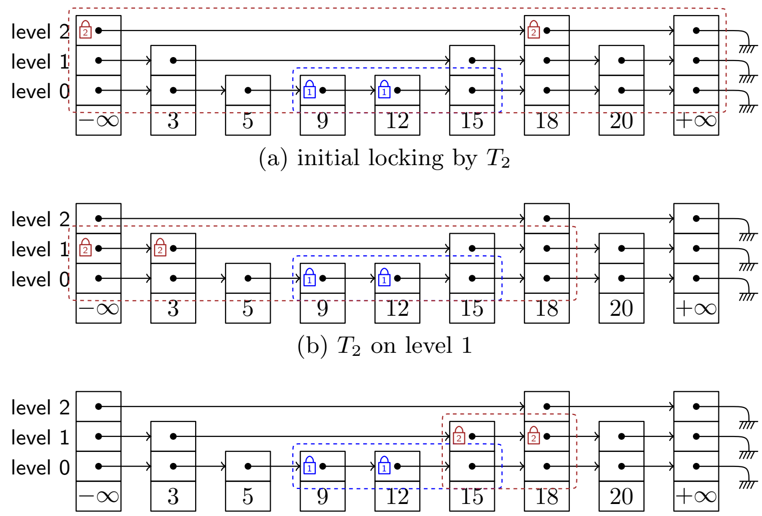


(c) T_1 on level 0

Fig. 15. Progress of T_1 towards insertion of value 14

We start executing T_1. This thread grabs the lock at level 2 on node $-\infty$ and 18, as shown in Fig. 15(a). As it detects that it has gone beyond the position where 16 should be inserted, it decides to go down a level. The algorithm proceeds as depicted in Fig. 15(b) and 15(c).

At this moment, T_2 starts its execution. Fig. 16 shown the progress made by T_2 toward the insertion of a level 2 node with value 16.



(a) initial locking by T_2

(b) T_2 on level 1

(c) final position of T_2 on level 1

Fig. 16. Progress of T_2 towards insertion of value 16

Notice that the potentially modifiable regions by thread T_1 and T_2 intersects, as shown in Fig. 16(c). In this case, it is quite easy to see that under the assumption of weak-fairness, if T_2 continuously perform the same insertion, it prevents T_1 from progressing. However, no problem exists under the assumption of strong fairness, since T_1 is not continuously enabled. Notice that T_1 becomes disable every time T_2 gets the lock at level 0 on node 15.

E Optimistic Lock-Coupling Skiplist

```
procedure INSERT(SkipList sl, Value newval)
    Vector⟨Node*⟩ upd[0..K − 1]                //@ mrgn m_r := ∅
                                              //@ L := sl.head
                                              //@ U := sl.tail
                                              //@ H := K − 1
    lvl := randomLevel(K)
    Node* curr := sl.head
    curr.locks[K − 1].lock()                  //@ m_r := m_r ∪ {(curr, K − 1)}
    Node* pred
    for i := K − 1 downto 0 do
        if i < K − 1 then
            curr.locks[i].lock()              //@ m_r := m_r ∪ {(curr, i)}
                                              //@ U := curr.next[i + 1]
            if i ≥ lvl then
                curr.locks[i + 1].unlock()    //@ m_r := m_r − {(curr, i + 1)}
                                              //@ H := i
            end if
        end if
        while curr.next[i].val < newval do
            curr.next[i].locks[i].lock()      //@ m_r := m_r ∪ {(curr.next, i)}
            pred := curr
            curr := curr.next[i]
            pred.locks[i].unlock()            //@ m_r := m_r − {(pred, i)}
                                              //@ if (i = lvl){L := curr}
        end while
        upd[i] := curr
    end for
    Bool valueWasIn := (curr.next[i].val = newval)
    if valueWasIn then
        for i := 0 to lvl do
            upd[i].locks[i].unlock()          //@ m_r := m_r − {(upd[i], i)}
        end for
    else
        x := CreateNode(lvl, newval)
        for i := 0 to lvl do
            x.next[i] := upd[i].next[i]
            upd[i].next[i] := x               //@ sl.r := sl.r ∪ {(x, i)}
            upd[i].locks[i].unlock()          //@ m_r := m_r − {(upd[i], i)}
        end for
    end if
                                              //@ H = −1
    return ¬valueWasIn
end procedure
```

Fig. 17. Optimistic algorithm for insertion on a concurrent lock-coupling skiplist

F Pessimistic Lock-Coupling Skiplist

```
procedure SEARCH(SkipList sl, Value v)
    int i := K − 1                          //@ mrgn m_r := ∅
                                            //@ L := sl.head
                                            //@ U := sl.tail
                                            //@ H := K − 1
    Node* pred := sl.head
    pred.locks[i].lock()                    //@ m_r := m_r ∪ {(pred, i)}
    Node* curr := pred.next[i]
    curr.locks[i].lock()                    //@ m_r := m_r ∪ {(curr, i)}
    while 0 ≤ i ∧ curr.val ≠ v do
        if i < K − 1 then
            pred.locks[i].lock()            //@ m_r := m_r ∪ {(pred, i)}
                                            //@ U := curr
            curr := pred.next[i]
            curr.locks[i].lock()            //@ m_r := m_r ∪ {(curr, i)}
            pred.next[i + 1].locks[i + 1].unlock()
                                            //@ m_r := m_r − {(pred.next[i + 1], i + 1)}
            pred.locks[i + 1].unlock()      //@ m_r := m_r − {(pred, i + 1)}
                                            //@ H := i
        end if
        while curr.val < v do
            pred.locks[i].unlock()          //@ m_r := m_r − {(pred, i)}
                                            //@ L := curr
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()            //@ m_r := m_r ∪ {(curr, i)}
        end while
        i := i − 1
    end while
    Bool valueIsIn := (curr.val = v)
    if i = K − 1 then
        curr.locks[i].unlock()              //@ m_r := m_r − {(curr, i)}
                                            //@ U := L
        pred.locks[i].unlock()              //@ m_r := m_r − {(pred, i)}
                                            //@ H := -1
    else
        curr.locks[i + 1].unlock()          //@ m_r := m_r − {(curr, i + 1)}
                                            //@ U := L
        pred.locks[i + 1].unlock()          //@ m_r := m_r − {(pred, i + 1)}
                                            //@ H := -1
    end if
    return valueIsIn
end procedure
```

Fig. 18. Pessimistic algorithm for searching on a concurrent lock-coupling skiplist

G Proving Termination of All Threads

In this section we describe how to extend our decision procedure in order to reason about the termination of all threads. We use the (L, U, H) regions introduced in Section C to represent the minimum portion of the skiplist we are sure a thread can potentially modify. We say that two threads conflict when the (L, U, H) of one is included into the (L, U, H) of the other one. The idea of the proof is that if a thread does not conflict with other thread, then it terminates. For such purpose we consider the optimistic and pessimistic version of the algorithms presented in Section E and F respectively.

To apply this idea, we need to define a $min : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \to \mathsf{setth}$ function which, given a memory representation and a skiplist regions denoted by a (L, U, H) triple, returns the set of thread identifiers whose (L, U, H) is contained in such region. To aid the definition of min, we extend $\mathsf{TSL_K}$ with a new function $skipRgTh : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \to \mathsf{setth}$, defined into Σ_{bridge}. The function $skipRgTh$ returns the set of thread identifiers that contain a lock on any node in the list that goes from a to b considering level l of the skiplist. Besides, we ask that the (L, U, H) of all thread identifiers in the set returned by $skipRgTh$ are contained into the region denoted by the L, U and H given as parameter.

Before we proceed with the definition of such function, there are some assumptions we need to make. In particular, we require that (L, U, H) values are not kept locally, as depicted in previous algorithms. We need them to be shared among all threads. Therefore, we assume that the *SkipList* class is extended with ghost mappings m_L, m_U and m_H which goes from thid to addr. These mappings are updated by the algorithms such that at every moment, for every thread identifier t, $(m_L(t), m_U(t), m_H(t))$ matches with the (L, U, H) of thread t.

We begin extending our decision procedure by adding a $cont$ predicate to $PATH$. This predicate holds when a region (L', U', H') is contained into a region (L, U, H):

$$
\begin{aligned}
cont\ &:\ \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \\
cont(h, L', U', H', L, U, H) &\mathrel{\hat{=}} reach_{\mathsf{K}}(h, L, L', 0) \quad \wedge \\
&\phantom{\mathrel{\hat{=}}}\ reach_{\mathsf{K}}(h, U', U, 0) \quad \wedge \\
&\phantom{\mathrel{\hat{=}}}\ (K' < K \vee K' = K)
\end{aligned}
$$

Basically, predicate $cont$ says that considering level 0 of the skiplist, from L it is possible to reach L', from U' it can be reached U and K' should be lower or equal to K. Of course, we can ensure that (L', U', H') represents a valid rectangular region by adding the constraint $reach_{\mathsf{K}}(h, L', U', 0)$ to the predicate. Then, we can extend $PATH$ further by adding a recursive $contTh$ predicate which takes as argument:

- a memory layout, h
- a lower bound address, L
- an upper bound address, U

– a bound on the skiplist's level, H
– an initial address, a
– a final address, b
– a level, l
– a set of thread identifiers, s

Then, the predicate $contTh(h, L, U, H, a, b, l, s)$ holds whether s is the set of threads identifiers owing a lock in the list that goes from address a to b, through level l of the skiplist. Moreover, we require that the (L, U, H) of each thread identifiers in s must be a subset of the region denoted by the L, U and K given as parameter. Formally, we define the predicate $contTh$ as:

$$contTh : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \times \mathsf{setth}$$

$$\begin{pmatrix} t = h[a].lockid[l] \wedge \\ t \neq \oslash \hfill \wedge \\ contained \end{pmatrix} \rightarrow contTh(h, L, U, H, a, a, l, \{t\}_T)$$

$$\begin{pmatrix} t = h[a].lockid[l] \hfill \wedge \\ (t = \oslash) \vee (t \neq \oslash \wedge \neg contained) \end{pmatrix} \rightarrow contTh(h, L, U, H, a, a, l, \emptyset_T)$$

$$\begin{pmatrix} t = h[a].lockid[l] \hfill \wedge \\ h[a].next[l] = a' \hfill \wedge \\ contTh(h, L, U, H, a', b, l, s) \wedge \\ t \neq \oslash \hfill \wedge \\ contained \end{pmatrix} \rightarrow contTh(h, L, U, H, a, b, l, s \cup_T \{t\}_T)$$

$$\begin{pmatrix} t = h[a].lockid[l] \hfill \wedge \\ h[a].next[l] = a' \hfill \wedge \\ contTh(h, L, U, H, a', b, l, s) \wedge \\ (t == null \vee \neg contained) \end{pmatrix} \rightarrow contTh(h, L, U, H, a, b, l, s)$$

where $contained \hat{=} cont(h, m_L(t), m_U(t), m_H(t), L, U, H)$

Notice that the definition of $contTh$ is similar to the one of $reach_{\mathsf{K}}$. Finally, we need to add to GAP the following equivalence:

$$contTh(h, L, U, H, a, b, l, s) \leftrightarrow skipRgTh(h, L, U, H, a, b, l) = s$$

We can now use function $skipRgTh$ to define the function min we required. We can do so through the following equivalence:

$$
\begin{aligned}
t = min(h, L, U, H) \leftrightarrow\ & s = skipRgTh(h, L, U, H) && \wedge \\
& t \in s && \wedge \\
& \emptyset = skipRgTh(h, m_L(t), m_U(t), m_H(t))
\end{aligned}
$$

As usual, every time we find a literal of the form $t = min(h, L, U, H)$ we proceed to replace it by the equivalent definition we have given above. Then, we replace the invocations of *skipRgTh* by invocations to *contTh*. We can finally unroll the occurrences of *contTh* according to its recursive definition up to the bound given by the small model property.

TODO: Remains to verify whether the SMP still holds.

```
procedure INSERT(SkipList sl, Value newval)
    Vector⟨Node*⟩ upd[0..K − 1]                  //@ mrgn m_r := ∅
                                                 //@ L := sl.head
                                                 //@ U := sl.tail
                                                 //@ H := K − 1
    lvl := randomLevel(K)
    Node* pred := sl.head
    pred.locks[K − 1].lock()                     //@ m_r := m_r ∪ {(pred, K − 1)}
    Node* curr := pred.next[K − 1]
    curr.locks[K − 1].lock()                     //@ m_r := m_r ∪ {(curr, K − 1)}
    Node* cover
    for i := K − 1 downto 0 do
        if i < K − 1 then
            pred.locks[i].lock()                 //@ m_r := m_r ∪ {(pred, i)}
                                                 //@ U := pred.next[i + 1]
            if i ≥ lvl then
                pred.locks[i + 1].unlock()       //@ m_r := m_r − {(pred, i + 1)}
                                                 //@ H := i
            end if
            if i < k − 2 ∧ i > lvl − 2 then
                cover.locks[i + 2].unlock()
            end if
            cover := curr
            curr := pred.next[i]
            curr.locks[i].lock()                 //@ m_r := m_r ∪ {(curr, i)}
        end if
        while curr.val < newval do
            pred.locks[i].unlock()               //@ m_r := m_r − {(pred, i)}
                                                 //@ if (i = lvl){L := curr}
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()                 //@ m_r := m_r ∪ {(curr, i)}
        end while
        upd[i] := pred
    end for
    Bool valueWasIn := (curr.val = newval)
    if valueWasIn then
        for i := 0 to lvl do
            upd[i].next[i].locks[i].unlock()     //@ m_r := m_r − {(upd[i].next[i], i)}
            upd[i].locks[i].unlock()             //@ m_r := m_r − {(upd[i], i)}
        end for
    else
        x := CreateNode(lvl, newval)
        for i := 0 to lvl do
            x.next[i] := upd[i].next[i]
            upd[i].next[i] := x                  //@ sl.r := sl.r ∪ {(x, i)}
            x.next[i].locks[i].unlock()          //@ m_r := m_r − {(x.next[i], i)}
            upd[i].locks[i].unlock()             //@ m_r := m_r − {(upd[i], i)}
        end for
    end if
                                                 //@ H = −1
    return ¬valueWasIn
end procedure
```

Fig. 19. Pessimistic algorithm for insertion on a concurrent lock-coupling skiplist

```
procedure Remove(SkipList sl, Value v)
    Vector < Node* > upd[0..K − 1]                //@ mrgn m_r := ∅
                                                  //@ L := sl.head
                                                  //@ U := sl.tail
                                                  //@ H := K − 1
    Node* pred := sl.head
    pred.locks[K − 1].lock()                      //@ m_r := m_r ∪ {(pred, K − 1)}
    Node* curr := pred.next[K − 1]
    curr.locks[K − 1].lock()                      //@ m_r := m_r ∪ {(curr, K − 1)}
    for i := K − 1 downto 0 do
        if i < K − 1 then
            pred.locks[i].lock()                  //@ m_r := m_r ∪ {(pred, i)}
                                                  //@ U := curr
            curr := pred.next[i]
            curr.locks[i].lock()                  //@ m_r := m_r ∪ {(curr, i)}
            if pred.next[i + 1].val ≠ e then
                pred.next[i + 1].locks[i + 1].unlock()
                pred.locks[i + 1].unlock()        //@ H := i
            end if
        end if
        while curr.val < v do
            pred.locks[i].unlock()                //@ m_r := m_r − {(pred, i)}
                                                  //@ if (aux.val ≠ e){L := curr}
            pred := curr
            curr := pred.next[i]
            curr.locks[i].lock()                  //@ m_r := m_r ∪ {(curr, i)}
        end while
        upd[i] := pred
    end for
    for i := K − 1 downto 0 do
        if upd[i].next[i] = curr ∧ curr.val = v then
            upd[i].next[i] := curr.next[i]        //@ sl.r := sl.r − {(curr, i)}
            curr.locks[i].unlock()                //@ m_r := m_r − {(curr, i)}
        else
            upd[i].next[i].locks[i].unlock()      //@ m_r := m_r − {upd[i].next[i], i)}
        end if
        upd[i].locks[i].unlock()                  //@ m_r := m_r − {(upd[i], i)}
                                                  //@ H := i − 1
    end for
    Bool valueWasIn := (curr.val = v)
    if valueWasIn then
        free (curr)
    end if
    return valueWasIn
end procedure
```

Fig. 20. Pessimistic algorithm for deletion on a concurrent lock-coupling skiplist